CompactRISC

CR16A
Programmer's Reference Manual

Part Number: 424521426-006

February 1997

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
1.0	August 1996	First release.
1.1	February 1997	Minor modifications and corrections.

PREFACE

This Programmer's Reference Manual presents the programming model for the CR16A microprocessor core. The key to system programming, and a full understanding of the characteristics and limitations of the CompactRISC Toolset, is understanding the programming model.

The information contained in this manual is for reference only and is subject to change without notice.

CONTENTS

FIGURES

TABLES

Chapter 1
INTRODUCTION

National Semiconductor's CompactRISC architecture is a RISC architecture specifically designed for embedded systems. CompactRISC technology features compact code generation, low power consumption, silicon-efficient implementations, the ability to tightly integrate on-chip acceleration, I/O and memory functions, and scalability from 8- to 64-bits.

CISC to RISC migration

The past decade has seen a gradual migration from CISC to RISC architectures. The silicon required to support CISC CPUs is too large, consumes too much power and is too expensive.

RISC architectures execute a small set of basic instructions to achieve higher performance with less silicon. This yields smaller, more economical silicon solutions that use less power, and provide higher performance. Higher frequency clock speeds further improve performance. Higher clock speeds require advanced concurrent processing techniques that allow simultaneous and overlapping execution of instructions. The most obvious of these is a multistage pipeline structure.

Modern RISC architectures are optimized for High Level Language (HLL) programs, usually written in C, and powerful, multiuser, multitasking operating systems such as UNIX and Windows. In these applications, processing speed is the most important consideration. As a result, modern RISC architectures are designed for “hyper” clock speeds with extensive, overlapping execution logic which yield highly optimized processors for workstation and fileserver applications.

However, existing RISC architectures are not well suited for embedded systems applications.

Embedded systems requirements

Embedded systems, being application specific, require a low system cost, while delivering sufficient performance for the application. They usually integrate on-chip memory and I/O with the processor. Clock frequency is limited by EMI and noise.

These system constraints dictate the requirements for embedded processors. In the past, general-purpose CISC processors were modified to serve this market. The resulting microcontroller architectures provided low-cost, highly-integrated systems, but at a cost. The architectures were convoluted, difficult to program using high-level languages, and not scalable from one data-path width to the next.

Embedded CISC microcontrollers were, however, successful because they met other important criteria of embedded systems:

- Low total system cost
- On-chip memory and I/O functions
- Small code size

Present RISC processors are optimized for use as computer CPUs, and are thus not a perfect fit for embedded systems. An emphasis on speed, and high-speed concurrent operation logic, yields large silicon implementations. Large off-chip memories and I/O require large on-chip bus interface units that further increase silicon size. Finally, the use of consistent fixed-width instructions simplifies the processor design, but increases code size, and the size of ROM, as compared with CISC architectures.

National's CompactRISC technology approach

Based on its experience with many high-volume designs using its previous 8-bit and 16-bit microcontroller and 32-bit CISC embedded processor families, National concluded that there was a need for a new class of high-performance embedded control systems. Such processors are suited for moving and shaping information applications such as wireless communication systems, multifunction peripherals, automotive subsystems, printers and fax machines, games, mass storage subsystems, communication switches and encryption devices. These applications require significant processing power and powerful development tools (for faster time to market), but must meet stringent cost and power consumption constraints.

To meet this need, National developed a new RISC processor technology that combines the advantages of RISC with the compact code generation of CISC, and is scalable from 8 bits to 64 bits.

In developing the CompactRISC technology, National had to rethink the process of traditional RISC architecture design. In analyzing RISC design against the goals of embedded systems design, CompactRISC architects found several ways to greatly reduce the amount of silicon required without significantly reducing the performance advantages of RISC.

CompactRISC Architecture Characteristics:

- Reduced concurrency logic
- Smaller bus interface units
- Smaller transistor count

We made a number of observations which affected the basic architectural decisions:

- Workstation-oriented RISC architectures trade silicon efficiency for instruction execution efficiency.
  Traditional RISC architectures squeeze every bit of execution time

from a fast system clock by using overlapping pipelines and highly parallel concurrent logic. However, this greatly increases both the number of on-chip transistors required, and the die size.
The performance needs of embedded controllers are quite different. There is less pressure to squeeze clock performance through every available design trick. CompactRISC eliminates a large amount of costly pipeline control logic just by reducing the number of pipeline stages from five, or more, to three. Shorter pipeline organization allows eliminating branch prediction mechanisms and bypass registers, while maintaining an acceptable overall performance level for embedded systems.

- Traditional RISC systems used large off-chip memory and I/O, requiring large and complex on-chip bus interface units. Embedded systems use on-chip memory wherever possible. Thus, the CompactRISC is designed with a minimum bus interface unit to on-chip memories. A controller for external memory accesses is added only when off-chip memory is needed. A separate peripheral bus controller is optimized for on-chip I/O. This approach yields maximum resource flexibility with minimum silicon overhead.

- Designs optimized for clock speeds of 100 MHz, or higher, required much greater margins in the basic transistor design. National realized that the overwhelming majority of embedded systems face EMI and noise problems that prohibit clock speeds in excess of 25 MHz, and designed the CompactRISC with transistors and signal paths optimized for clock speeds of 30 MHz, and lower.

These observations led to a less complex, and much tighter, silicon implementation. Because the CompactRISC core has fewer transistors than other RISC processors, there are fewer transistors to drive, reducing the size of internal bus drivers. The cumulative effect of fewer and smaller transistors, with a slower clock, yields dramatically smaller silicon, with lower system costs, and significantly lower power consumption.

Reduced memory requirements

RISC architectures have traditionally used fixed-width instructions to simplify instruction decoder design. In 32-bit RISC systems, instructions are either four or eight bytes. CISC systems use a variable instruction length, resulting in smaller code size for a given application. The CompactRISC uses variable instruction widths, with fixed coding fields within the instruction itself. For example, the opcode field is always in the first 16 bits, with additional bytes as required for immediate values. Instructions for the 32-bit CompactRISC core may be 2-bytes, 4-bytes or 6-bytes long, but basic instructions are only two bytes long. This permits optimized instruction processing by the instruction decoder, and results in a smaller code size. The size of code generated for the CompactRISC core is comparable to CISC code size, or typically 25 percent smaller than code generated for a typical RISC CPU.

Standard 32-bit RISC processors deliver high performance only when aligned 32-bit data is used. Intermediate results are stored in memory as 32-bit values and registers are saved as 32-bit operands on the stack. CompactRISC CR32 instructions operate on 8-, 16- and 32-bit data. Nonaligned accesses are allowed. Dedicated data type conversion instructions speed data access to mixed size data. With smaller code size, and variable length instructions and data, the CompactRISC family makes more efficient use of smaller, lower cost, lower bandwidth memories. Smaller memories allow many more system elements to be integrated with on-chip memory.

Scalable cores from 8 to 64 bits

These architectural features make the CompactRISC technology ideal for the next generation of embedded systems. In addition, National decided to implement the CompactRISC technology in a set of core processors, in the range 8- to 64-bits. This provides a new, more attractive solution for designers of low-end embedded systems.

Low-cost, single-chip systems, with on-board memory and I/O, could be implemented with low-cost 8-bit microcontrollers, or later 16-bit versions. However, the accumulator-based architectures of such microcontrollers made programming in high-level languages impractical. Code generated by a compiler for such machines is typically larger, and slower than code written in assembly language.

The CompactRISC's architecture, however, produces highly efficient compiler generated code. CompactRISC is the first architecture designed to produce highly efficient HLL-generated code in both the 8- and 16-bit worlds.

With National's compatible family of processor cores, the designer of embedded controller-based systems can now choose the optimum processor size for a given target application. This is particularly useful in leveraging the development investment across several classes of related end-products. With a single processor family, a number of different products can be developed with a single development platform and using the same HLL-based development and debug tools.

Fast HLL development and debug

With the CompactRISC technology, unlike previous microcontroller families, you can develop code, and debug, in C, greatly speeding up the development process. In today's marketplace, with some product life cycles lasting 9 to 18 months or less, working with HLL "power tools" is a significant advantage. A single development platform, for a wide range of systems from low-end to high-end, is also an important advantage. The designer of CompactRISC-based systems can maintain a single set of software development tools covering all CompactRISC implementations.

CompactRISC architecture

In many ways, CompactRISC technology is a traditional RISC load/store processor architecture. For example, the CR16 executes an optimized instruction set with 21 internal registers grouped in 16 general purpose registers, three dedicated address registers, a processor status register and a configuration register. A three-stage pipeline is used to obtain a peak performance of 30 Million Instruction Per Second (MIPS) at a clock frequency of 30 MHz.

The CompactRISC core includes a pipelined integer unit that supports a peak execution speed of one instruction per each internal cycle, with a 60 Mbyte/sec. (CR16) or 120 Mbyte/sec. (CR32) pipelined bus. The CompactRISC technology supports little-endian memory addressing. This means that the byte order in a CompactRISC processor is from least significant byte to the most significant byte.

Debug support

The CompactRISC provides instruction tracing, soft breakpoints via breakpoint instructions and external ISE support.

You use instruction tracing, during debugging, to single-step through a program. Two bits in the PSR or Program Status Register enable and generate trace traps. In addition, you can use a breakpoint instruction (EXCP BPT) to stop execution of a program at specified instructions, allowing the debugger to examine the status of the program and internal registers. The CompactRISC also supports an input pin that causes an ISE interrupt. The core processor then provides status signals that are activated upon completing an instruction. Finally, the CR32 also has on-chip hardware breakpoint support for data and address values.

1.1 MANUAL ORGANIZATION

This reference manual describes the CR16A core, as follows:

Chapter 1 "INTRODUCTION" introduces the CompactRISC architecture.

Chapter 2 "PROGRAMMING MODEL"presents the instruction set, the register set, operands and addressing modes of the core.

Chapter 3 "EXCEPTIONS" introduces traps and interrupts in the core, and describes the way these exceptions are treated.

Chapter 4 "ADDITIONAL TOPICS" covers more advanced concepts related to the CR16A programming model: the CR16A debugging features and the use of the CR16A pipeline.

Chapter 5 "INSTRUCTION SET" describes all the instructions provided by the CR16A core.

Appendix A "INSTRUCTION SET ENCODING"

1.2 REFERENCES

Core Bus Specification., National Semiconductor, December 1994.

1.3 THE CORE

The CR16A core is designed to operate with other Very Large Scale Integrated (VLSI) modules such as a bus interface unit, ROM, RAM, and peripherals. Together with such modules, a CR16A based integrated circuit, implements a full microcontroller or microprocessor.

The CR16A is designed to obtain the maximum performance from Very Large Scale Integration (VLSI), pipelining, and optimizing compilers. The high performance of the CR16A microprocessor results from the implementation of a pipelined architecture with state-of-the-art VLSI CMOS technology.

The CR16A supports a peak execution speed of one instruction each clock cycle and a two byte/cycle pipelined system bus.

Chapter 2
PROGRAMMING MODEL

2.1 DATA TYPES

Integer Data Type

The integer data type is used to represent integers. Integers may be signed or unsigned. Two integer sizes are supported: 8-bit (1 byte), and 16-bit (1 word). Signed integers are represented as binary two's complement numbers and have values in the range -2^7 to 2^7-1 and -2^{15} to $2^{15}-1$, respectively. Unsigned numbers have values in the range 0 to 2^8-1 and 0 to $2^{16}-1$, respectively.

Boolean Data Type

The boolean data type is represented as an integer (byte or word). The value of its least significant bit represents one of two logical values, true or false. Integer 1 indicates true; integer 0 indicates false.

2.2 INSTRUCTION SET

This section includes a summary list of all the instructions in the CR16A instruction set. Chapter 5, "INSTRUCTION SET" describes each instruction in detail.

The following instructions are included in the CR16A:

Mnemonic	Operands	Description
MOVES		
MOVi	Rsrc/imm, Rdest	Move
MOVXB	Rsrc, Rdest	Move with sign extension
MOVZB	Rsrc, Rdest	Move with zero extension
INTEGER ARITHMETIC		
ADD[U]i	Rsrc/imm, Rdest	Add
ADDCi	Rsrc/imm, Rdest	Add with carry
MULi	Rsrc/imm, Rdest	Multiply
SUBi	Rsrc/imm, Rdest	Subtract (Rdest := Rdest − Rsrc)
SUBCi	Rsrc/imm, Rdest	Subtract with carry (Rdest := Rdest − Rsrc − PSR.C)
INTEGER COMPARISON		
CMPi	Rsrc/imm, Rdest	Compare (Rdest − Rsrc)

Mnemonic	Operands	Description
LOGICAL AND BOOLEAN		
ANDi	Rsrc/imm, Rdest	Logical AND
ORi	Rsrc/imm, Rdest	Logical OR
Scond	Rdest	Save condition code as boolean
XORi	Rsrc/imm, Rdest	Logical exclusive OR
SHIFTS		
ASHUi	Rsrc/imm, Rdest	Arithmetic left/right shift
LSHi	Rsrc/imm, Rdest	Logical left/right shift
BITS		
TBIT	Roffset/imm, Rsrc	Test bit
PROCESSOR REGISTER MANIPULATION		
LPR	Rsrc, Rproc	Load processor register
SPR	Rproc, Rdest	Store processor register
JUMPS AND LINKAGE		
Bcond	disp	Conditional branch
BAL	Rlink, disp	Branch and link
BR	disp	Branch
EXCP	vector	Trap (vector)
Jcond	Rtarget	Conditional Jump
JAL	Rlink, Rtarget	Jump and link
JUMP	Rtarget	Jump
RETX		Return from exception
LOAD AND STORE		
LOADi	disp(Rbase), Rdest	Load (register relative)
	disp(Rpair+1, Rpair), Rdest	Load (far-relative)
	abs, Rdest	Load (absolute)
STORi	Rsrc, disp(Rbase)	Store (register relative)
	Rsrc, disp(Rpair +1, Rpair)	Store (far-relative)
	Rsrc, abs	Store (absolute)
MISCELLANEOUS		
DI		Disable maskable interrupts
EI		Enable maskable interrupts
NOP		No operation
WAIT		Wait for interrupt

2.3 REGISTER SET

This section describes each register, its bits and its fields in detail. In addition, the format of each register is illustrated.

All registers are 16 bits wide, except for the three address registers, which are 18 bits wide. Bits specified as “reserved” must be written as 0. Read operations return a value of “undefined” from reserved bits.

The CR16A has 21 internal registers grouped by function as follows:

- 16 general purpose registers
- the following processor registers,
  - 3 dedicated address registers
  - 1 processor status register
  - 1 configuration register

Figure 2-1 shows the internal registers of the CR16A.

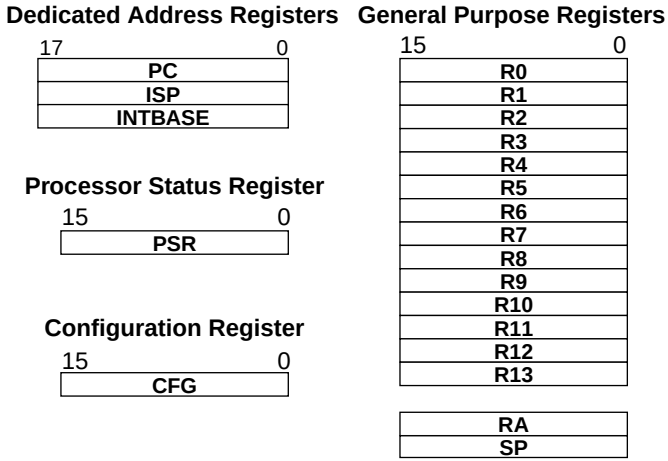


Figure 2-1. CR16A Registers

2.3.1 General Purpose Registers

Registers R0-R13 are used for general purposes, such as holding variables, addresses or index values. The SP general purpose register is usually used as a pointer to the program run-time stack. The RA general purpose register is usually used as a return address from sub-routines. If a general purpose register is specified by an operation that is 8 bits long, then only the low part of the register is used; the high part is not referenced or modified.

2.3.2 Dedicated Address Registers

This section describes the three 18-bit wide dedicated address registers that the CR16A uses to implement specific address functions.

PC Program Counter – The value in the PC register, points to the first byte of the instruction currently being executed. The most significant and the least significant bits of the PC are always 0. Thus CR16A instructions reside in even addresses in the address range 0 to $1FFFE_{16}$. At reset, the PC is initialized to 0 and the value of bits 1 through 16 of the PC prior to reset is saved in the R0 general purpose register.

ISP Interrupt Stack Pointer – The ISP register points to the lowest address of the last item stored on the interrupt stack. This stack is used when interrupt and trap service procedures are invoked. The two most significant, and the least significant bits of this register are always 0. Thus the interrupt stack always starts at an even address, and resides in the address range 0 to $FFFE_{16}$.

INTBASE Interrupt Base Register – The INTBASE register holds the address of the dispatch table for interrupts and traps. Refer to Chapter 3, "EXCEPTIONS" for more information. The least significant bit and the two most significant bits of this register are always 0. Thus the dispatch table always starts at an even address, and resides in the address range 0 to $FFFE_{16}$.

2.3.3 The Processor Status Register

The Processor Status Register (PSR) holds status information and selects operating modes for the CR16A. It is 16 bits wide. Figure 2-2 shows the format of the PSR.

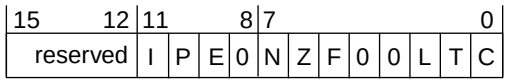


Figure 2-2. Processor Status Register

At reset, bits 0 through 11 of the PSR are cleared to 0, except for the PSR.E bit, which is set to 1. In addition, the value of each bit prior to reset is saved in the R1 general purpose register.

Several bits in the PSR have a dedicated condition code in the Conditional Branch (**Bcond**) instruction. These bits are Z, C, L, N, and F. Any Bcond instruction can cause a branch in the program execution, basd on the value of one of these PSR bits, or a combination of them. For example, one of the **Bcond** instructions, **BEQ** (Branch EQual), causes a branch if the PSR.Z flag is set. Refer to the **Bcond** instruction in "Instruction Definitions" on page 5-1 for details.

Bits 3, 4 and 8 have a constant value of 0. Bits 12 through 15 of the PSR register are reserved. The other bits are described below:

The C Bit The Carry bit indicates whether a carry or borrow occurred after addition or subtraction. It can be used with the **ADDC** and **SUBC** instructions to perform multiple-precision integer arithmetic calculations. It is cleared to 0 if no carry or borrow occurred, and set to 1 if a carry or borrow occurred.

The T Bit The Trace bit causes program tracing. While the T bit is set to 1, a Trace (TRC) trap is executed after every instruction. Refer to "Instruction Tracing" on page 4-1 for more information on program tracing. The T bit is automatically cleared to 0, when a trap or an interrupt occurs. The T bit is used in conjunction with the P bit, see below.

The L Bit The Low flag is set by comparison operations. In integer comparison, the L flag is set to 1, if the second operand (Rdest) is less than the first operand (Rsrc) when both operands are interpreted as unsigned integers. Otherwise, it is cleared to 0. Refer to the specific compare instruction in "Instruction Definitions" on page 5-1 for details.

The F Bit The Flag bit is a general condition flag which is set by various instructions. It may be used to signal exceptional conditions or to distinguish the results of an instruction (e.g., integer arithmetic instructions use it to indicate overflow from addition or subtraction). In addition it is set, or cleared, as a result of a Test-Bit instruction.

The Z Bit The Zero bit is set by comparison operations. In integer comparisons it is set to 1 if the two operands are equal. Otherwise, it is cleared to 0. Refer to the specific compare instruction in "Instruction Definitions" on page 5-1 for details.

The N Bit The Negative bit is set by comparison operations. In integer comparison it is set to 1 if the second operand (Rdest) is less than the first operand (Rsrc) when both operands are interpreted as signed integers. Otherwise it is cleared to 0. Refer to the specific compare instruction in “Instruction Definitions” on page 5-1 for details.

The E Bit The local maskable interrupt Enable bit affects the state of maskable interrupts. While this bit and the PSR.I bit are 1, all maskable interrupts are accepted. While this bit is 0, only the non-maskable interrupt is accepted. On reset the E bit is set to 1. See “Interrupt Handling” on page 3-3.

There are two dedicated instructions that set and clear the E bit. It is set to 1 by the Enable Interrupts instruction (EI). It is cleared to 0 by the Disable Interrupts instruction (DI). This pair can be used to locally disable maskable interrupts, regardless of the global state of maskable interrupts, which is determined by the value of the PSR.I bit.

See also “Interrupt Handling” on page 3-3.

The P Bit The Trace (TRC) trap Pending bit is used together with the T bit to prevent a TRC trap from occurring more than once for any instruction. It may be cleared to 0 (no TRC trap pending) or 1 (TRC trap pending). See “Exception Service Procedures” on page 3-6 and “Instruction Tracing” on page 4-1 for more information.

The I Bit The global maskable Interrupt enable bit affects the state of maskable interrupts. While this bit and the PSR.E bits are 1, all maskable interrupts are accepted. While this bit is 0, only the non-maskable interrupt is accepted. The I bit is cleared to 0 on reset. In addition, it is automatically cleared when an interrupt occurs.

2.3.4 The Configuration Register

The Configuration Register (CFG) is used to enable or disable various operating modes and to control optional on-chip caches in some cores that are based on the CompactRISC technology. The CR16A does not support any optional modules, and all the bits of the CFG register are reserved.

Figure 2-3 shows the format of the CFG register.

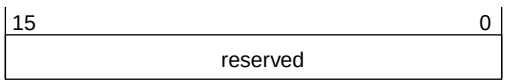


Figure 2-3. Configuration Register

2.4 MEMORY ORGANIZATION

The CR16A implements 18-bit addresses. This allows the CPU to access up to 256 Kbytes of data, and 128 Kbytes of program memory. The memory is a uniform linear address space. Memory locations are numbered sequentially starting at 0 and ending at $2^{18}-1$. The number specifying a memory location is called an address.

The contents of each memory location is a byte consisting of eight bits. Instructions and data can occupy any byte address in the range of 0 through 128 Kbyte and 256 Kbyte respectively, except for addresses $0FC00_{16}$ through $0FFFF_{16}$ which are reserved, (see Figure 2-4).

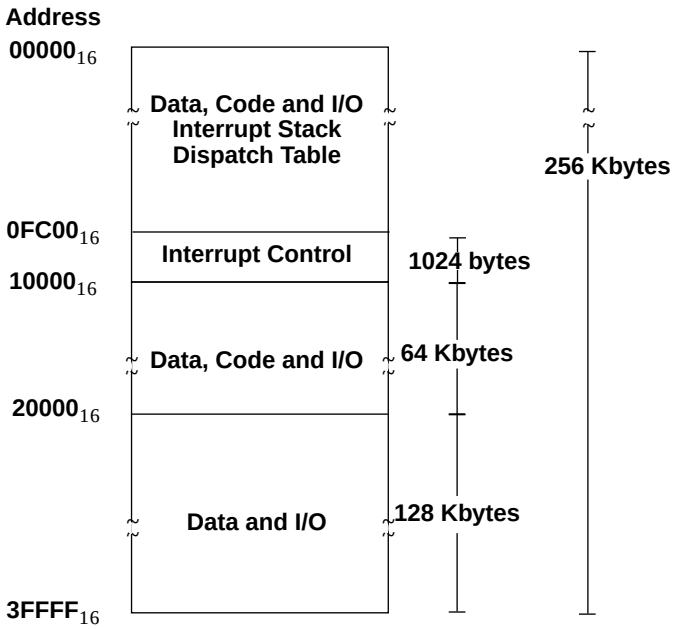


Figure 2-4. Memory Organization

2.4.1 Data References

Bit and byte order for data references

The CR16A refers to bytes, and words, of data in memory. When it refers to a byte of data in memory, the data is actually located at the specified address. When it refers to a word of data in memory, the data is located at consecutive bytes beginning with the byte at the specified address.

The byte order in the CR16A is from least significant to most significant byte (little-endian). The address of a word of data refers to the least significant byte of the data value; the remaining byte is located at a higher address.

Bits are ordered from least significant to most significant. The least significant bit is in position zero. The **TBIT** instruction refers to bits by their ordinal position numbers. Figure 2-5 shows the memory representation for data values.

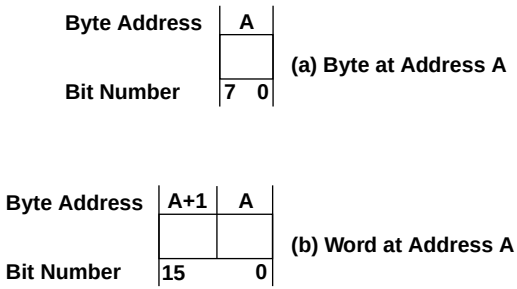


Figure 2-5. Data Representation

Data references

The CR16A supports references to memory by the load and store instructions. Bytes, and words can be referenced on any boundary.

2.4.2 Stacks

A *stack* is a one-dimensional data structure in which values are entered and removed, one item at a time, at one end called the *top-of-stack*. It consists of a block of memory and a variable called the *stack pointer*. Stacks are important data structures in both systems and applications programming. They are used to store status information during subroutine calls and interrupt servicing. Also, algorithms for expression evaluation in compilers and interpreters depend on stacks to store intermediate results. High level languages, such as C, keep local data and other information on a stack.

The CR16A provides support for two kinds of stacks: the interrupt stack and the program stack.

The interrupt stack

The processor uses the interrupt stack to save and restore the program state during the handling of an exception condition. This information is automatically pushed, by the hardware, on to the interrupt stack before entering an exception service procedure. On exit from the exception service procedure, the hardware pops this information from the interrupt stack. See Chapter 3, "EXCEPTIONS" for more information on this subject. The interrupt stack can reside in the first 64 Kbyte of the address range, and is accessed via the ISP processor register.

The program stack

The program stack is normally used by programs at run time, to save and restore register values upon procedure entry and exit. It is also used to store local and temporary variables. The program stack is accessed via the SP general-purpose register, and therefore must reside in the first 64 Kbyte of the address range. Note that this stack is handled by software only, e.g., the CompactRISC C Compiler generates code that pushes data on to, and pops data from, the program stack.

Both stacks expand downward in memory, toward address zero.

2.5 ADDRESSING MODES

Most instructions use one, two or three of the CR16A's registers as operands. Some instructions may also use an immediate value instead of the first register operand. Memory is accessed only by the load and store instructions, which use absolute, relative or far-relative addressing mode.

The following addressing modes are available:

Register mode

In register mode, the operand is located in a general purpose register, i.e., R0 through R13, RA or SP. The following instruction illustrates register addressing mode.

```
ADDB R1, R2
```

Immediate mode

In immediate mode, the operand is a constant value which is specified within the instruction. For example:

```
MULW $4, R4
```

PC-Relative mode

In PC-Relative mode, the operand is a displacement from the current value of the PC register. For example:

```
BR *+10
```

Relative mode In relative mode, the operand is located in memory. Its address is obtained by adding the contents of a general purpose register to the constant value in the displacement field encoded in the instruction. The following instruction illustrates relative addressing mode.

```
LOADW 12(R5), R6
```

Far-relative mode In far-relative mode, the operand is located in memory. Its address is obtained by concatenating a pair of adjacent general purpose registers to form an 18-bit value, and adding this value to the constant value in the displacement field encoded in the instruction.

The 16 least significant bits of the 18-bit value are taken from the base register, and the two most significant bits of the value are taken from the two least significant bits in the next consecutive register. The following instruction illustrates far-relative addressing mode.

```
STORW R7, 4(R3, R2)
```

Absolute mode In absolute mode, the operand is located in memory and its address is specified within the instruction. The following example illustrates absolute addressing mode.

```
LOADB 4000, R6
```

Chapter 3
EXCEPTIONS

3.1 INTRODUCTION

Program *exceptions* are conditions which alter the normal sequence of instruction execution, causing the processor to suspend the current process, and execute a special service procedure, often called a handler.

Interrupts

An exception resulting from the activity of a source external to the processor is known as an *interrupt*; an exception which is initiated by some action or condition in the program itself is called a *trap*. Thus, an interrupt need have no relationship to the executing program, while a trap is caused by the executing program and will recur each time the program is executed. The CR16A recognizes nine exceptions: six traps and three types of interrupts.

The exception handling technique employed by an interrupt-driven processor determines how fast the processor can perform input/output transfers, the speed with which transfers between tasks and processes can be achieved, and the software overhead required for both. Therefore, it determines to a large extent the efficiency of a processor's multiprogramming and multitasking (including real-time) capabilities.

Addressing interrupts

Exception handling in the CR16A uses a Dispatch Table in the first 64-Kbyte of the memory whose base address is contained in the Interrupt Base register (INTBASE). See Figure 3-1.

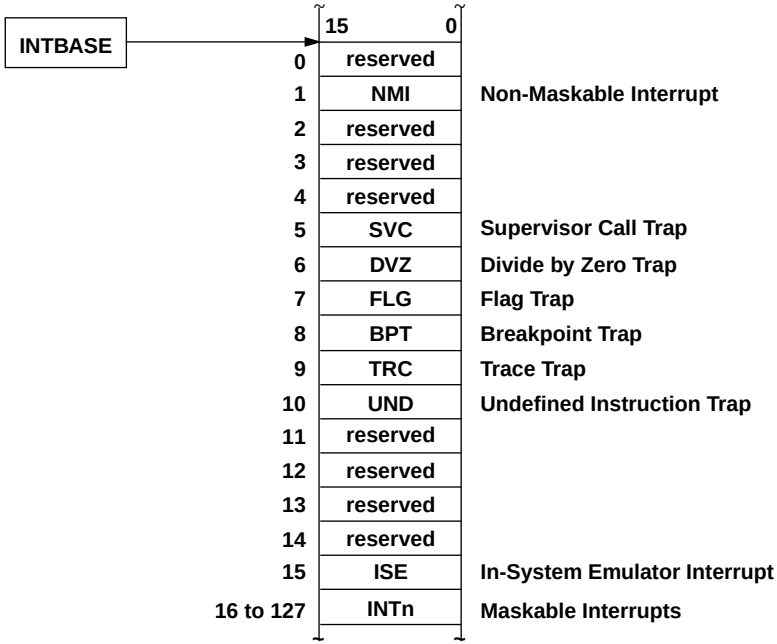


Figure 3-1. Dispatch Table

For purposes of addressing the Dispatch Table, each of the exceptions has been assigned a number. This exception number (or interrupt vector) is used to compute the starting address of the service procedure for the particular exception required, i.e., the exception number is multiplied by two, and added to the contents of the Interrupt Base register (INTBASE). The resulting value matches the entry in the Dispatch Table that provides bits 1 through 16 of the address of the exception service procedure. The processor reconstructs the full address of the exception service procedure using the fact that its LSB (as is the LSB of any CR16A instruction address) is always 0.

The interrupt process

When an exception occurs, the CPU automatically preserves the complete machine state of the program immediately prior to the occurrence of the exception. A copy of the PC and the PSR is made and pushed onto the Interrupt Stack. Depending on the kind of exception, it restores and/or adjust the contents of the Program Counter (PC) and the Processor Status Register (PSR). The interrupt exception number is then used to obtain the address of the exception service procedure from the dispatch table, which is then called.

The **RETX** instruction returns control to the interrupted program, and restores the contents of the PSR and the PC registers to their previous status. See the **RETX** instruction on page 5-28.

The following two subsections describe interrupts and traps in detail.

3.1.1 Interrupt Handling

Interrupt handling in the CR16A provides a number of features which contribute to efficiency and programming flexibility. For example, rather than saving all registers when an interrupt occurs, the CR16A automatically saves only the Program Counter (PC) and the Program Status Register (PSR); the other registers are under program control. They may be saved and restored by the interrupt handlers. This provides a high degree of flexibility in adjusting interrupt response speed, and facilitates context switching for interrupts.

An Interrupt Stack allows context switching in a multiprogramming or multitasking environment without disabling interrupts.

The CR16A provides three types of interrupts: maskable interrupts, non-maskable interrupt (NMI) and In-System Emulator (ISE).

Maskable interrupts

Maskable interrupts are disabled whenever PSR.E or PSR.I are cleared to 0. PSR.I serves as the global interrupt mask, while PSR.E serves as a local interrupt mask. PSR.E can be easily changed by using the **EI** and **DI** instructions (see the **EI** instruction on page 5-13 and the **DI** instruction on page 5-12). PSR.E should be used when read-modify-write must be an atomic operations (i.e. no interrupt should occur between the read and the write).

Upon receipt of a maskable interrupt, the processor determines the vector number by performing an interrupt acknowledge bus cycle in which a byte is read from address $0FE00_{16}$. This byte contains a number in the range 16-127, which is used as an index into the Dispatch Table to find the address of the appropriate interrupt handler. Then, control is transferred to that interrupt handler.

Non-maskable interrupt

Non-maskable interrupts cannot be disabled; they occur when catastrophic events (such as an imminent power failure) require immediate handling to preserve system integrity. Non-maskable interrupts use vector number 1 in the Dispatch Table. When a non-maskable interrupt is detected, the CR16A performs an interrupt-acknowledge bus cycle to address $0FF00_{16}$, and discards the byte that is read during the bus cycle.

ISE interrupt

In-System Emulator (ISE) interrupts cannot be disabled; they temporarily suspend execution when an appropriate signal is activated. ISE interrupts use vector number 15 in the Dispatch Table. When an ISE interrupt is detected, the CR16A performs an interrupt-acknowledge bus cycle to address $0FC00_{16}$, and discards the byte that is read during the bus cycle.

3.1.2 Traps

The CR16A recognizes the following traps:

BPT Trap **Breakpoint Trap** - Used for program debugging. Caused by the **EXCP BPT** instruction.

SVC Trap **Supervisor Call Trap** - Temporarily, transfers control to supervisor software, typically to access facilities provided by the operating system. Caused by the **EXCP SVC** instruction.

FLG Trap **Flag Trap** - Indicates various computational exceptional conditions. Caused by the **EXCP FLG** instruction.

DVZ Trap **Division by Zero Trap** - Indicates an integer division by zero. Caused by the **EXCP DVZ** instruction, which can be used by integer division emulation code to indicate this exception.

UND Trap **Undefined Instruction Trap** - Indicates undefined op codes. Caused by an **EXCP UND** instruction or an attempt to execute any of the following:

- any undefined instruction;
- the **EXCP** instruction when a reserved field in the dispatch table is specified.

TRC Trap **Trace Trap -** A TRC trap occurs before an instruction is executed when the PSR.P bit is 1. Used for program debugging and tracing. See Chapter 4, “ADDITIONAL TOPICS” for more information.

3.2 EXCEPTION PROCESSING

3.2.1 Instruction Endings

The CR16A checks for exceptions at various points during the execution of instructions. Some exceptions, such as interrupts, are acknowledged between instructions, i.e., before the next instruction is executed. Other exceptions, such as a Division by Zero (DVZ) trap, are acknowledged during execution of an instruction. In such a case, the instruction is suspended. See Table 3-1.

When an instruction is suspended, it is not completed, but all other previously issued instructions have been completed. Result operands and flags (except for the PSR.P bit on some traps) are not affected. In this case, the PC saved on the interrupt stack contains the address of the suspended instruction.

When an interrupt is detected while a **MULi** instruction is being executed, the **MULi** instruction is suspended.

3.2.2 Acknowledging an Exception

The CR16A performs the following operations in response to interrupt or trap exceptions:

1. Decrements the Interrupt Stack Pointer (ISP) by 4.

2. Saves the contents of the current PSR and of the current value of bits 1 through 16 of the PC on the interrupt stack. See Figure 3-2. The contents of the PSR are located at the higher address.

3. Alters the PSR by clearing certain control bits. See Table 3-1.

4. For interrupts, displays information during the interrupt acknowledge bus cycle to indicate the type of interrupt encountered.
   If the interrupt is a maskable interrupt, the CPU reads the vector number during this cycle from address $FE00_{16}$, that is mapped to the Interrupt Control Unit (ICU).
   If the interrupt is a Non Maskable Interrupt (NMI) the CPU performs a read operation from address $FF00_{16}$ for observability purposes. If the interrupt is an ISE interrupt the CPU also performs a read operation from address $FC00_{16}$ for observability purposes.

5. Reads the word entry from the dispatch table at address (INTBASE) + vector × 2. The dispatch table entry is used to call the exception service procedure and is interpreted as a pointer that is loaded into bits 1 through 16 of the PC. Bits 0 and 17 of the PC are cleared. See Figure 3-3.

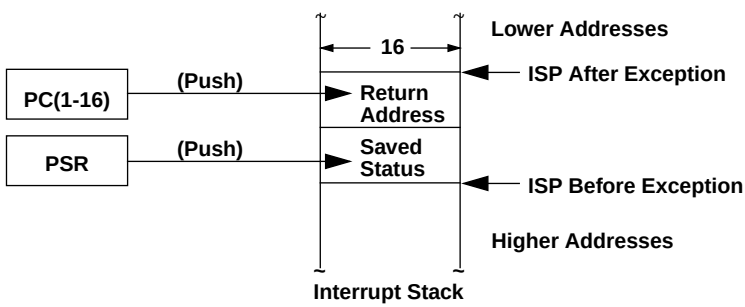


Figure 3-2. Saving the PC and PSR Contents During an Exception Acknowledge Sequence

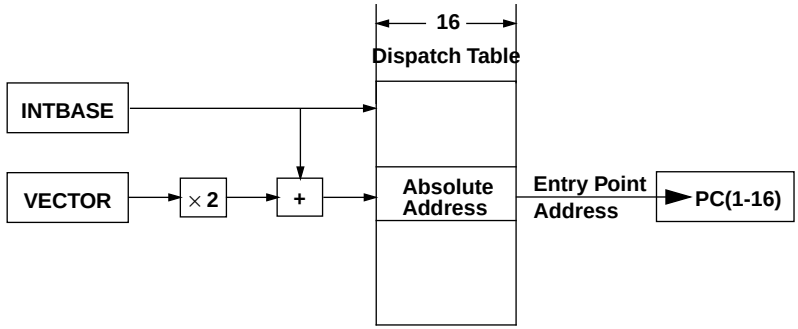


Figure 3-3. Transfer of Control During an Exception Acknowledge Sequence

Table 3-1 summarizes how each type of exception is acknowledged.

Table 3-1. Summary of Exception Processing

Exception	Instruction Completion Status	PC Saved	Cleared PSR Bits	
			Before Saving PSR	After Saving PSR
Interrupt	Before start of instruction	Next	None	I P T
Interrupt during execution of **MULi**	Suspended	Current	P[a]	I P T
BPT, DVZ, FLG, SVC	Suspended	Current	None	P T
UND	Suspended	Current	None	P T
TRC	Before start of instruction	Next	P	P T

a. The PSR.P bit is cleared when an interrupt is acknowledged before a **MULi** instruction is completed, to prevent a mid-instruction trace trap upon return from the exception service procedure.

3.2.3 Exception Service Procedures

After the CR16A acknowledges an exception, control is transferred to the appropriate exception service procedure. The TRC trap is disabled (the PSR.P and PSR.T bits are cleared). Maskable interrupts are also disabled (the PSR.I bit is cleared) for a service procedure called in response to an interrupt.

At the beginning of each instruction, the PSR.T bit is copied into the PSR.P. If PSR.P is still set at the end of the instruction, a TRC trap is executed before the next instruction.

To complete a suspended instruction, the exception service procedure should be programmed to do one of the following:

Simulate a suspended instruction

The exception service procedure can use software to simulate execution of the suspended instruction. After it calculates and writes the results of the suspended instruction, it should modify the flags in the copy of the PSR which were saved on the interrupt stack, and update the PC saved on the interrupt stack to point to the next instruction to be executed.

The exception service procedure can then execute the **RETX** instruction, and the CR16A will begin executing the instruction following the suspended instruction. For example, when an Undefined Instruction Trap (UND) occurs, software can be used to perform the appropriate corrective actions.

Retry execution of a suspended instruction

The suspended instruction can be retried after the exception service procedure has corrected the trap condition that caused the suspension.

In this case, the exception service procedure should execute the **RETX** instruction at its conclusion; then the CR16A will retry the suspended instruction. A debugger takes this action when it encounters an **EXCP BPT** instruction that was temporarily placed in another instruction's location in order to set a breakpoint. In this case, exception service procedures should clear the PSR.P bit to prevent a TRC trap from occurring again.

3.2.4 Returning From Exception Service Procedures

Service procedures perform actions appropriate for the type of exception detected. At their conclusion, service procedures execute the **RETX** instruction to resume executing instructions at the point where the exception was detected.

3.2.5 Priority Among Exceptions

The CR16A checks for specific exceptions at various points while executing an instruction (see Figure 3-4).

If several exceptions occur simultaneously, the CR16A responds to the exception with the highest priority.

If several maskable interrupts occur simultaneously, the Interrupt Control Unit (ICU) determines the highest priority interrupt, and requests the CR16A to service this interrupt.

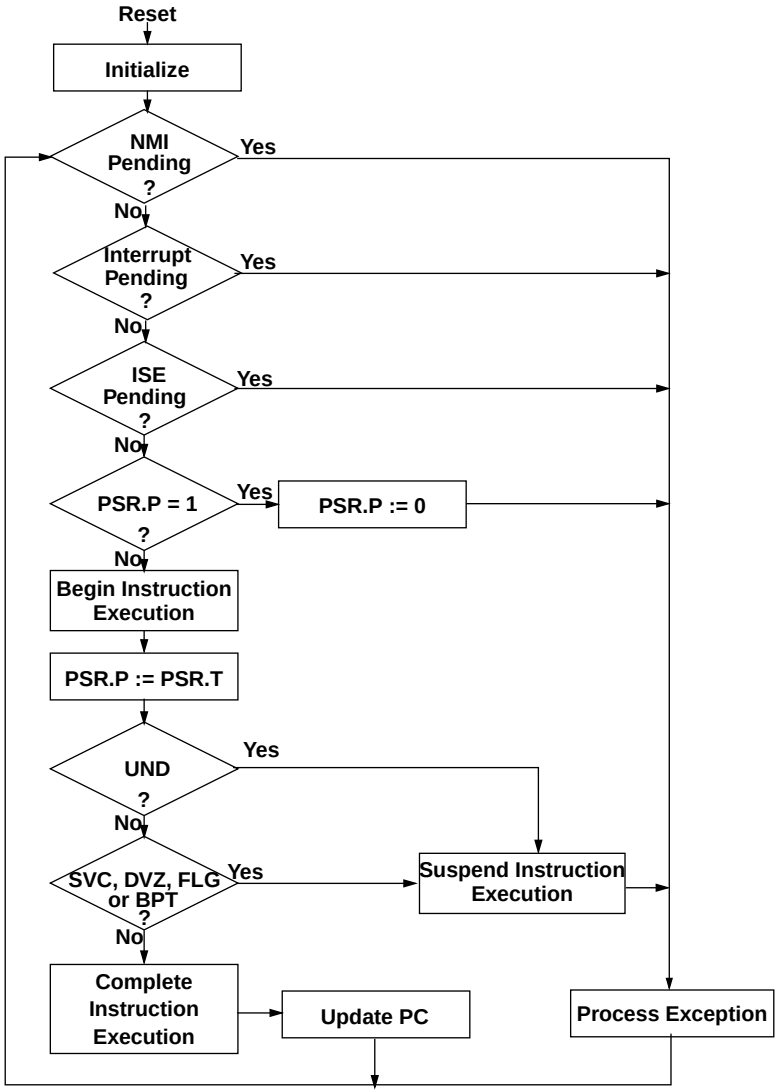


Figure 3-4. Exception Processing Flowchart

Before executing an instruction

Before executing an instruction, the CR16A checks for pending interrupts and trace traps, in that order. It responds to the interrupts in order of descending priority (i.e., first non-maskable interrupts, then maskable interrupts and lastly, ISE interrupts.

If no interrupt is pending, and PSR.P is 1 (i.e., a trace trap is pending), then the CR16A clears PSR.P and processes the trace trap.

If no trace trap or interrupt is pending, then the CR16A begins executing the instruction by copying PSR.T to PSR.P. While executing an instruction, the CR16A may detect a trap.

During execution of an instruction

First, the CR16A checks for an undefined instruction (UND) trap; then it looks for any of the following mutually exclusive traps: SVC, DVZ, FLG or BPT. The CR16A responds to the first trap it detects by suspending the current instruction and executing the trap.

If an undefined instruction is detected, then no data references are performed for the instruction.

If an interrupt becomes pending during execution of the **MULi** instruction the CR16A clears PSR.P to 0 and responds to the requested interrupt.

If no exception is detected while the instruction is executing, then the instruction is completed (i.e., values are changed in registers and memory, except for PSR.P, which was changed earlier) and the PC is updated to point to the next instruction.

3.2.6 Nested Interrupts

A nested interrupt is an interrupt that occurs while another interrupt is being serviced. Since the PSR.I bit is automatically cleared before any interrupt is serviced (see Table 3-1), nested maskable interrupts are not serviced by default. However, the Exception Service Procedure can explicitly allow nested maskable interrupts at any point, by setting the PSR.I bit using a LPR instruction. In this case, pending maskable interrupts are serviced normally even in the middle of the currently executing Exception Service Procedure.

It is possible to enable nesting of specific maskable interrupts inside a certain Exception Service Procedure. This is done by programming the Interrupt Control Unit (ICU) to mask the undesired interrupt sources, during the execution of the Exception Service Procedure. This should be done before the PSR.I bit is set.

Nested Non Maskable Interrupt (NMI) and nested ISE interrupt are always serviced.

The interrupt nesting level is limited only by the amount of memory that is available for the interrupt stack.

3.3 RESET

A reset occurs when the appropriate signal is activated. Reset must be used at power-up to initialize the CR16A.

As a result of a reset operation:

- All instructions currently being executed are terminated.
- Results and flags normally affected by the terminated instruction are unpredictable.
- The results of instructions, whose execution started but did not yet end, may not be written to their destinations.
- Any pending interrupts and traps are eliminated.

Upon reset, the following operations are executed:

1. The current values of bits 1 through 16 of the PC are stored in R0, and the current value of the PSR is stored in R1.

2. The following internal registers are cleared to 0: PC, CFG and PSR, except for PSR.E, which is set to 1.

After reset, the processor begins normal execution at memory location 0, and the reserved bits in these registers, and the contents of all other registers, are unpredictable.

Chapter 4
ADDITIONAL TOPICS

This chapter discusses debugging support and instruction execution order.

4.1 DEBUGGING SUPPORT

The CR16A provides the following features to make program debugging easier.

- Instruction Tracing
- Soft Break Generation by Breakpoint Instruction
- ISE Support

The PSR is used to control these features.

4.1.1 Instruction Tracing

Instruction tracing can be used during debugging to single-step through selected portions of a program. The CR16A uses two bits in the PSR to enable and generate trace traps. Tracing is enabled by setting the T bit in the PSR register.

During the execution of each instruction, the CR16A copies the PSR.T bit into the PSR.P (trace pending) bit. Before beginning the next instruction, the CR16A checks the PSR.P bit to determine whether a Trace (TRC) trap is pending. If PSR.P is 1, i.e., a trace trap is pending, the CR16A generates a trace trap before executing the instruction.

If any other trap or interrupt is requested during execution of a traced instruction, its entire service procedure is allowed to complete before the TRC trap occurs.

For example, if an Undefined Instruction (UND) trap is detected while tracing is enabled, the TRC trap occurs after execution of the **RETX** instruction that marks the end of the UND service procedure. The UND service procedure can use the PC value saved on top of the interrupt stack to determine the location of the instruction. The UND service procedure is not affected, whether instruction tracing was enabled or not.

Each interrupt and trap sequence can update the PSR.P bit, when required for proper tracing. This guarantees only one TRC trap per instruction, and that the return address pushed during a TRC trap is always the address of the next instruction to be traced.

Note the following:

- **LPR** (on PSR) and **RETX** instructions cannot be reliably traced because they may alter the PSR.P bit during their execution.
- If instruction tracing is enabled while the **WAIT** instruction is executed, a trace trap occurs after the next interrupt, when the interrupt service procedure returns.

The Breakpoint Instruction

The breakpoint instruction (**EXCP BPT**) may be used by debuggers to stop the execution of a program at specified instructions, to examine the status of the program. The debugger replaces these instructions with the breakpoint instruction. It then starts the program execution. When such an instruction is reached, the breakpoint instruction causes a trap, which enables the debugger to examine the status of the program at that point.

4.1.2 In-System Emulator (ISE) Support

The CR16A core provides the following to support the development of real-time In-System Emulator (ISE) equipment and Application Development Boards (ADBs).

- Status signals that indicate when an instruction in the execution pipeline is completed and the length of this instruction.
- Status signals that indicate the type of each bus cycle, e.g., fetch.
- Status signals that indicate when there is a non-sequential fetch.
- An ISE interrupt signal.
- An interrupt acknowledge cycle.
- A special bus status signal during exception handling, that indicates that the dispatch table is being read.
- Upon reset, the CR16A stores the contents of the PSR in R1 and the contents of bits 1 through 16 of the PC in R0.

4.2 INSTRUCTION EXECUTION ORDER

The CR16A has four operating states in which instructions may be executed and exceptions may be processed. They are:

- Reset
- Executing Instructions
- Processing Exception
- Waiting for Interrupt

These states and the transitions between them are shown in Figure 4-1.

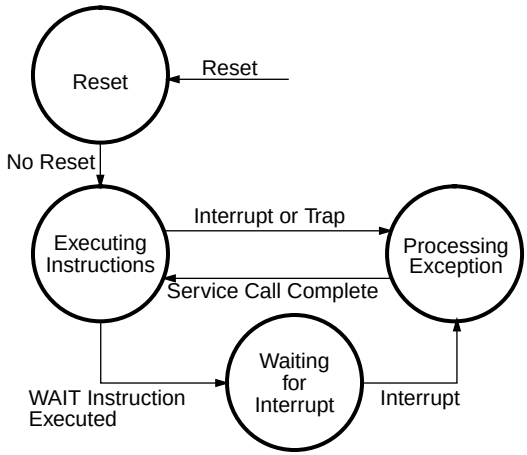


Figure 4-1. CR16A Operating States

Reset When the reset input signal is activated, the CR16A enters the reset state. In the reset state, the contents of certain dedicated registers are initialized, as specified in “Reset” on page 3-10.

Executing Instructions When the reset signal is deactivated the CR16A enters the executing-instructions state. In this state, the CR16A executes instructions repeatedly until an exception is recognized or a **`WAIT`** instruction is executed.

Processing Exception When an exception is recognized the CR16A enters the processing exception state in which it saves the PC and the PSR contents. The processor then reads an absolute address from the Interrupt Dispatch Table and branches to the appropriate exception service procedure. Refer to “The interrupt process” on page 3-2 for more information.

To process maskable interrupts, the CR16A also reads a vector value from an Interrupt Control Unit (ICU).

After successfully completing all data references required to process an exception, the CR16A reverts to the executing instructions state.

Waiting for Interrupt

When a **WAIT** instruction is executed, the CR16A enters the wait for interrupt state in which it is idle. When an interrupt is detected the processor enters the processing exception state.

4.2.1 The Instruction Pipeline

The operations for each instruction are not necessarily completed before the operations of the next instruction begin. The CR16A can overlap operations for several instructions, using a pipelined technique to enhance its performance. While the CR16A is fetching one instruction, it can simultaneously be decoding a second instruction and calculating results for a third instruction. See Figure 4-2.

In most cases, pipelined instruction execution improves performance while producing the same results as strict sequential instruction execution. Under certain circumstances, however, the effects of this performance enhancement are visible to system software and hardware as differences in the order of memory references performed by the CR16A. See explanation below.

Instruction Fetches

The CR16A fetches an instruction only after all previous instructions have been completely fetched. But, it may begin fetching the instruction before all of the source operands have been read, and before results have been written for previous instructions.

Operands and Memory References

The source operands for an instruction are read only after all data reads, and data writes in previous instructions have been completed. This process and the order of precedence of memory reference for two consecutive instructions is illustrated in Figure 4-2. The arrows indicate order of precedence between operations in an instruction and between instructions.

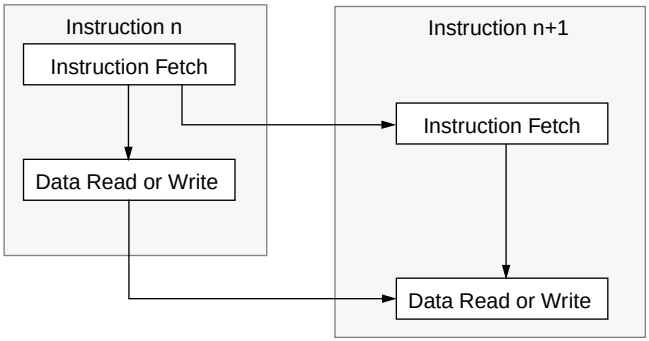


Figure 4-2. Memory References for Consecutive Instructions

Overlapping Operations As a consequence of overlapping the operations for several instructions, the CR16A may fetch an instruction, but not execute it (for example, if the previous instruction causes a trap). The CR16A reads source operands and writes destination operands for executed instructions only.

Dependencies The CR16A does not check for dependencies between the fetching of the next instruction and the writing of the results of the previous instructions. Therefore, special care is required when executing self-modifying code.

4.2.2 Serializing Operations

The CR16A serializes instruction execution after processing an exception. This means that it finishes writing all the results of the preceding instructions to a destination, before it fetches the next instruction. This fetch is non-sequential.

The CR16A also serializes instruction execution after executing the following instructions: **LPR**, **RETX**, and **EXCP**.

Chapter 5
INSTRUCTION SET

This chapter describes each of the CR16A instructions, in detail.

5.1 INSTRUCTION DEFINITIONS

The name of each operand appears in bold italics, and indicates its use. In addition, the valid addressing modes, access class and length are specified for each operand. The addressing mode may be: *reg* (register), *procreg* (processor register), *imm* (immediate), *abs* (absolute), *rel* (relative) or *far* (far relative). The access class may be *read*, *write*, *rmw* (read-modify-write), *addr* (address) or *disp* (displacement). The access class is followed by a data length attribute specifier. See Figure 5-1.

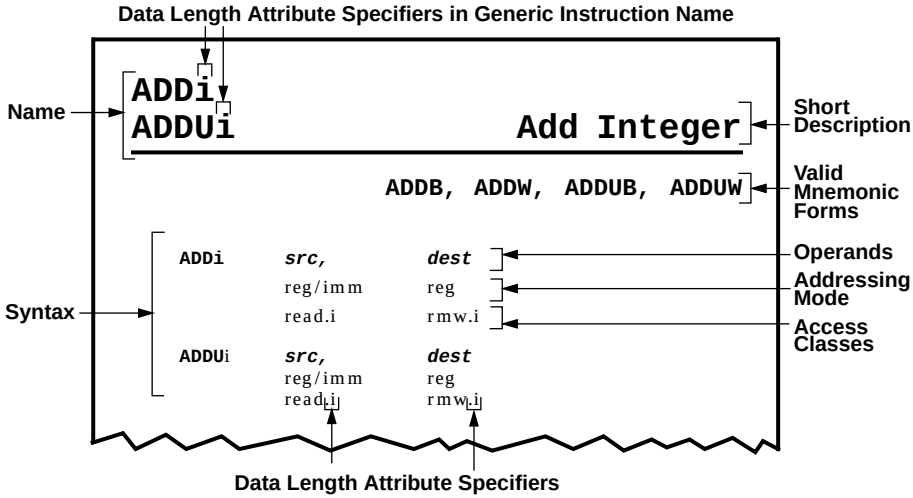


Figure 5-1. Instruction Header Format

The data length attribute specifier specifies how the operands will be interpreted, and represents a character that is incorporated into the name of the actual instruction. The **i** specifier stands for a **B** (byte) or **W** (word) in the actual instruction name.

Each instruction definition is followed by a detailed example of one or more typical forms of the instruction. In each example, all the operands of the instruction are identified, both those explicitly stated in assembly language and those that will be implicitly affected by the instruction.

For each example, the values of operands before and after execution of the instruction are shown. Often the value of an operand is not changed by the instruction. When the value of an operand changes, it's field is highlighted, i.e., its box is grey. See Figure 5-2.

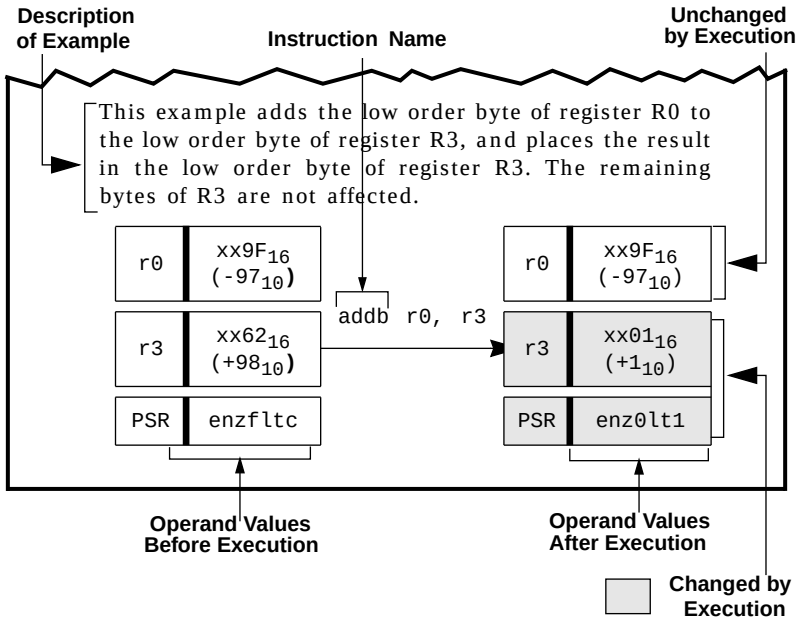


Figure 5-2. Instruction Example Format

The least significant digit of the least significant byte is the rightmost digit. Values are expressed in terms of a radix in a subscript to the value.

An **x** represents a binary digit or a hexadecimal digit (4 bits) that is either ignored or unchanged.

5.2 DETAILED INSTRUCTION LIST

The following pages describe in detail the instruction set.

ADDi
ADDUi — Add Integer

ADDB, ADDW, ADDUB, ADDUW

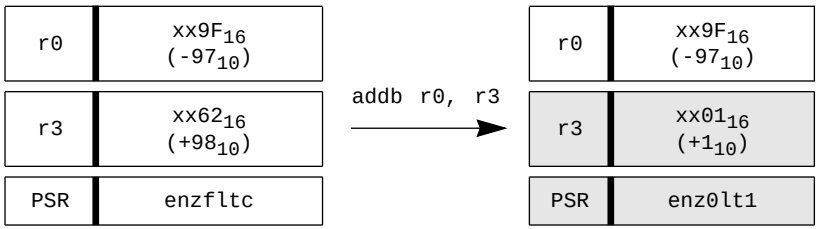
ADDi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i
ADDUi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **ADDi** and **ADDUi** instructions add the ***src*** and ***dest*** operands, and place the result in the ***dest*** operand.

Flags: During execution of an **ADDi** instruction, PSR.C is set to 1 on a carry from addition, and cleared to 0 if there is no carry. PSR.F is set to 1 on an overflow from addition, and cleared to 0 if there is no overflow. PSR flags are not affected by the **ADDUi** instruction.

Traps: None

Example: This example adds the low order byte of register R0 to the low order byte of register R3, and places the result in the low order byte of register R3. The remaining bytes of R3 are not affected.



ADDCi — Add Integer with Carry

ADDCB, ADDCW

ADDCi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

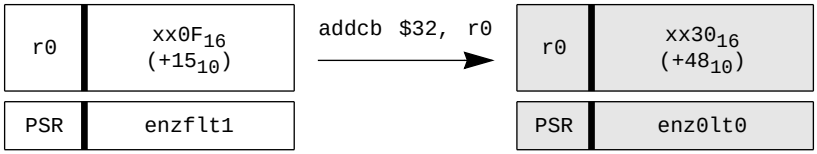
The **ADDCi** instructions add the ***src*** operand ***dest*** operand and the PSR.C flag, and places the sum in the ***dest*** operand.

Flags: PSR.C is set to 1 if a carry occurs, and cleared to 0 if there is no carry. PSR.F is set to 1 if an overflow occurs, and cleared to 0 if there is no overflow.

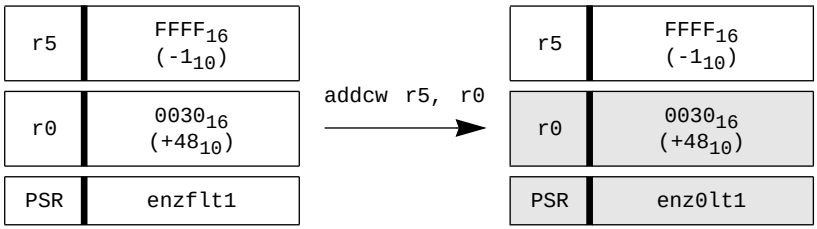
Traps: None

Examples:

1. Example 1 adds 32, the low order byte of register R0, and the PSR.C flag contents and places the result in the low order byte of register R0. The remaining bytes of register R0 are unaffected.



2. Example 2 adds the contents of registers R5 and R0, and the contents of the PSR.C flag, and places the result in register R0.



ANDi — Bitwise Logical AND

ANDB, ANDW

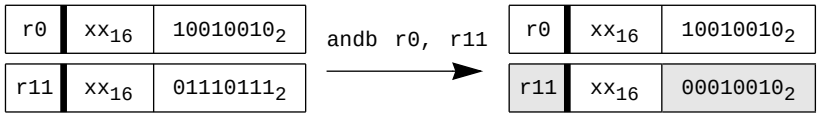
ANDi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **ANDi** instructions perform a bitwise logical AND operation on the ***src*** and dest operands, and places the result in the ***dest*** operand.

Flags: None

Traps: None

Example: This example ANDs the low order bytes of registers R0 and R11 and places the result in the low order byte of register R11. The remaining byte of register R11 is unaffected.



ASHUB, ASHUW

ASHUi	***count***,	***dest***
	reg/imm	reg
	read.B	rmw.i

The **ASHUi** instructions perform an arithmetic shift on the ***dest*** operand as specified by the ***count*** operand. Both operands are interpreted as signed integers.

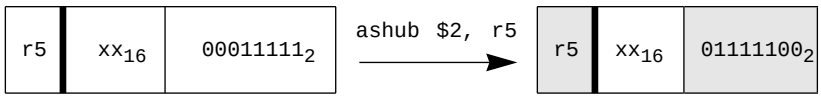
The sign of ***count*** determines the direction of the shift. A positive ***count*** specifies a shift to the left; a negative ***count*** specifies a shift to the right. The absolute value of the ***count*** specifies the number of bit positions to shift the ***dest*** operand. The ***count*** operand value must be in the range –7 to +7 if **ASHUB** is used; and in the range –15 to +15 if **ASHUW** is used. Otherwise, the result is unpredictable.

If the shift is to the left, high order bits (including the sign bit) shifted out of ***dest*** are lost, and low order bits emptied by the shift are filled with zeros. If the shift is to the right, low order bits shifted out of ***dest*** are lost, and high order bits emptied by the shift are filled from the original sign bit of ***dest***.

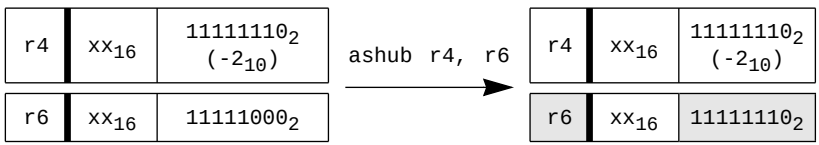
Flags: None

Traps: None

Examples: 1. Example 1 shifts the low order byte of register R5 two bit positions to the left. The remaining byte of register R5 is unaffected.



2. Example 2 reads a byte from register R4. Based on this value, it shifts the low order byte of register R6 accordingly. The remaining byte of register R6 is unaffected.



Bcond — Conditional Branch

BEQ, BNE, BCS, BCC, BHI, BLS, BGT, BLE, BFS, BFC, BLO, BHS, BLT, BGE

Bcond ***dest***
imm
disp

If the condition specified by **cond** is true, the **Bcond** instruction causes a branch in program execution. Program execution continues at the location specified by ***dest***, sign extended to 18 bits, plus the current contents of the Program Counter. Both the least significant bit and the most significant bit of the address are cleared to 0. If the condition is false, execution continues with the next sequential instruction.

cond is a two-character condition code that describes the state of a flag or flags in the PSR. If the flag(s) are set as required by the specified **cond**, the condition is true; otherwise, the condition is false. The following table describes the possible **cond** codes and the related PSR flag settings:

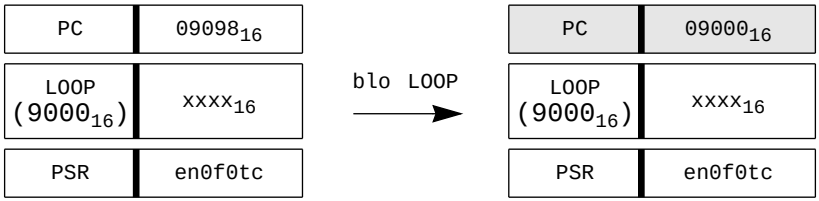
cond Code	**Condition**	**True State**
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal	N flag is 0
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag is 0
LO	Lower	Z and L flags are 0
HS	Higher or Same	Z or L flag is 1
LT	Less Than	Z and N flags are 0
GE	Greater Than or Equal	Z or N flag is 1

Flags: None

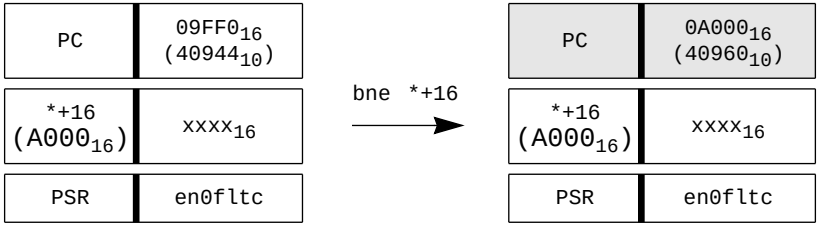
Traps: None

Examples:

1. Example 1 passes execution control to the instruction labeled LOOP by adding $1FF68_{16}$ to the PC, if the PSR.Z and PSR.L flags are 0.



2. Example 2 passes execution control to a non-sequential instruction if the PSR.Z flag is 0. The instruction passes execution control by adding 16 to the PC register.



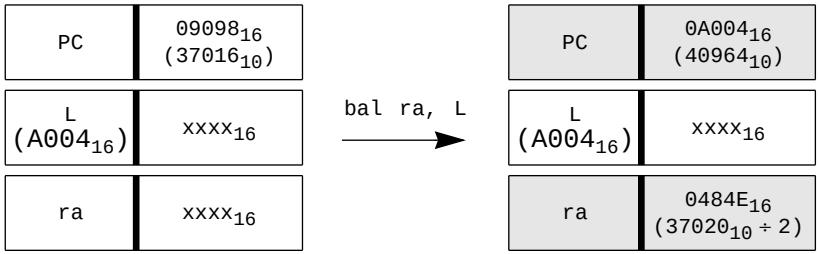
BAL	***link***,	***dest***
	reg	imm
	write.W	disp

The address of the next sequential instruction is first stored in the register specified as the ***link*** operand. Then, program execution continues at the address specified by ***dest***, sign extended to 18 bits, plus the current contents of the PC register. Both the least significant bit and the most significant bit of the address are cleared to 0.

Flags: None

Traps: None

Example: This example saves bits 1 through 16 of the PC of the next sequential instruction in register RA, and passes execution control to the instruction labeled L by adding $00F6C_{16}$ to the current PC.



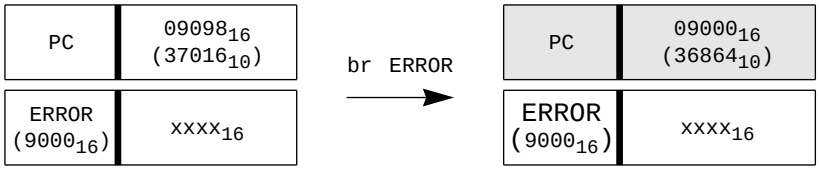
BR ***dest***
imm
disp

dest is sign extended to 18 bits and added to the current contents of the PC register. The most and least significant bits of the PC are cleared to 0. The result is loaded into the PC. Program execution continues at the location specified by the updated PC.

Flags: None

Traps: None

Example: This example passes execution control to the instruction labeled ***ERROR*** by adding $1FF68_{16}$ to the PC.



CMPi — Compare Integer

CMPB, CMPW

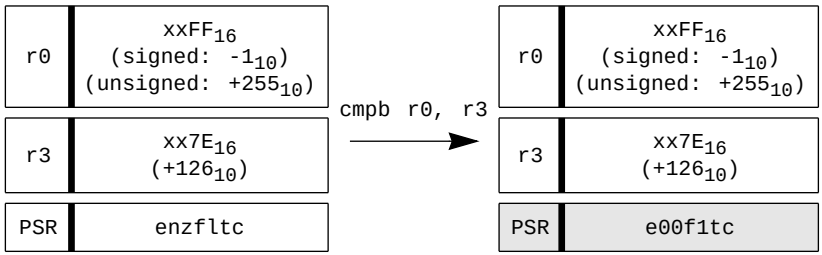
CMPi	***src1***,	***src2***
	reg/imm	reg
	read.i	read.i

The **CMPi** instruction subtracts the ***src1*** operand from the ***src2*** operand, and sets the PSR.Z, PSR.N, and PSR.L flags to indicate the comparison result. The PSR.N flag indicates the result of a signed integer comparison; the PSR.L flag indicates the result of an unsigned comparison. Both types of comparison are performed.

Flags: PSR.Z is set to 1 if ***src1*** equals ***src2***; otherwise it is cleared to 0. PSR.N is set to 1 if ***src1*** is greater than ***src2*** (signed comparison); otherwise it is cleared to 0. PSR.L is set to 1 if ***src1*** is greater than ***src2*** (unsigned comparison); otherwise it is cleared to 0.

Traps: None

Example: The following example compares low order bytes in registers R0 and R3.



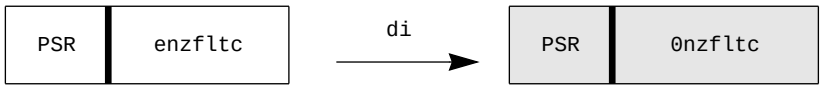
```
DI
```

The **DI** instruction clears PSR.E to 0. Maskable interrupts are disabled regardless of the value of PSR.I.

Flags: PSR.E is cleared to 0.

Traps: None.

Example: The following example clears the PSR.E bit.



EI

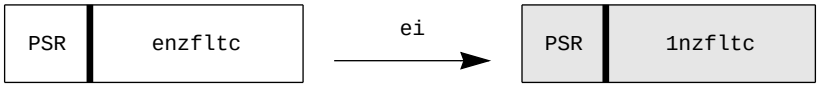
EI

The **EI** instruction sets PSR.E to 1. If PSR.I is also 1, maskable interrupts are enabled.

Flags: PSR.E is set to 1.

Traps: None

Example: The following example sets the PSR.E bit to 1.



EXCP ***vector***

The **EXCP** instruction activates the trap specified by the ***vector*** operand. The return address pushed onto the interrupt stack is the address of the **EXCP** instruction itself.

Flags: None

Traps: The traps that will occur are determined by the value of the ***vector*** operand as shown in the following table.

Vector	Trap Name
SVC	Supervisor Call
DVZ	Division by Zero
FLG	Flag
BPT	Breakpoint
UND	Undefined Instruction
otherwise	*reserved*

Example: This example activates the Supervisor Call Trap.

```
excp svc
```

Flags:

Traps:

Example:

Jcond Conditional Jump

JEQ, JNE, JCS, JCC, JHI, JLS, JGT, JLE, JFS, JFC, JLO, JHS, JLT, JGE

Jcond ***dest***
reg
addr.W

If the condition specified by **cond** is true, the **Jcond** instruction causes a jump in program execution. Program execution continues at the address specified in the ***dest*** register, by loading its contents into bits 1 through 16 of the PC register. Bits 0 and 17 of the PC are cleared to 0. If the condition is false, execution continues with the next sequential instruction.

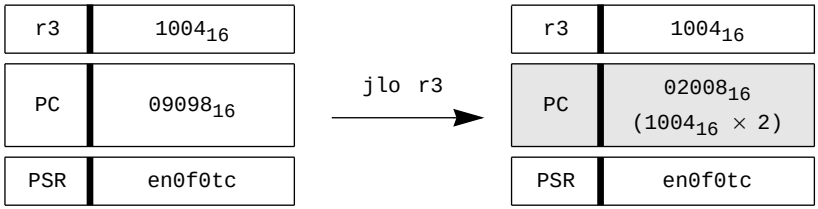
cond is a two-character condition code that describes the state of a flag or flags in the PSR. If the flag(s) are set as required by the specified **cond**, the condition is true; otherwise, the condition is false. The following table describes the possible **cond** codes and the related PSR flag settings:

cond Code	**Condition**	**True State**
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal	N flag is 0
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag is 0
LO	Lower	Z and L flags are 0
HS	Higher or Same	Z or L flag is 1
LT	Less Than	Z and N flags are 0
GE	Greater Than or Equal	Z or N flag is 1

Flags: None

Traps: None

Example: In this example, the CR16A loads the address held in R3 into bits 1 through 16 of the PC register, and program execution continues at that address, if the PSR.Z and PSR.L flags are 0.



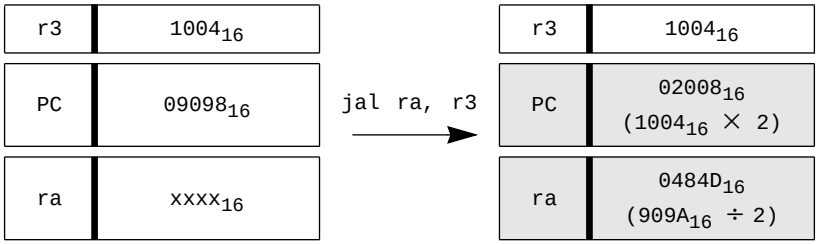
JAL	***link***,	***dest***
	reg,	reg
	write.W	addr.W

Program execution continues at the address specified in the ***dest*** register, by loading its contents into bits 1 through 16 of the PC register. Bits 0 and 17 of the PC register are cleared to 0.Bits 1 through 16 of the address of the next sequential instruction are stored in the register specified by the ***link*** operand.

Flags: None

Traps: None

Example: This example loads the address held in R3 into the bits 1 through 16 of the PC register. Program execution continues at that address. Bits 1 through 16 of the address of the next sequential instruction are stored in register RA.



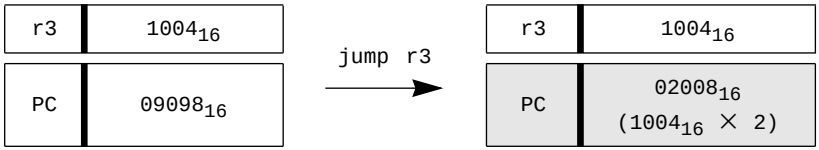
JUMP ***dest***
reg
addr.W

Program execution continues at the address specified in the ***dest*** register, by loading its contents into bits 1 through 16 of the PC register, and clearing the least and most significant bits to 0.

Flags: None

Traps: None

Example: This example loads the address held in R3 into bits 1 through 16 of the PC register. Program execution continues at that address.



LOADi — Load Integer

LOADB, LOADW

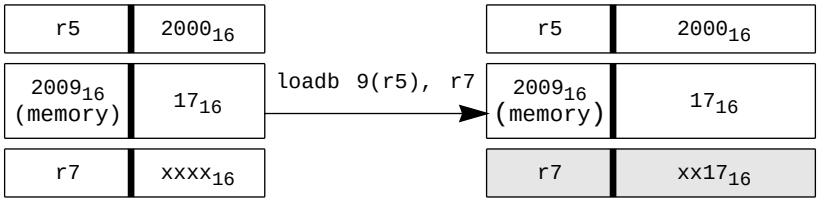
LOADi	***src***,	***dest***
	abs/rel/far	reg
	read.i	write.i

The **LOADi** instructions load the ***src*** operand from memory, and places it in the ***dest*** register operand.

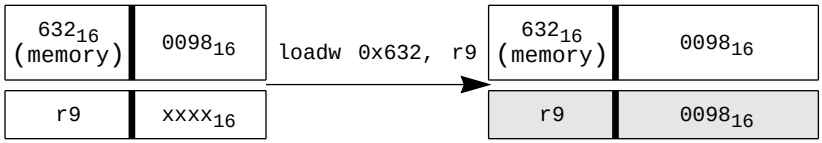
Flags: None

Traps: None

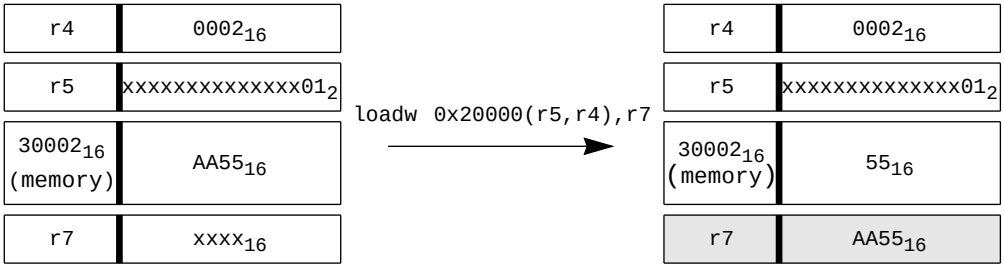
Examples: 1. Example 1 loads a byte operand in address 9(R5) to the low order byte of register R7. The remaining byte of register R7 is unaffected.



2. Example 2 loads a word operand in address 632 to register R9.



3. Example 3 loads a word operand in address 30002_{16} to register R7. The address is formed by adding 20000_{16} to the value in R4 concatenated with the value in R5.



LPR	*src*,	*dest*
	reg	procreg
	read.W	write.W

The **LPR** instruction copies the ***src*** operand to the processor register specified by ***dest***.

If ***dest*** is ISP or INTBASE, only bits 0 through 15 are written, and the least significant bit (bit 0) and the two most significant bits (bits 16,17) of the address are cleared to 0.

The following processor registers may be loaded:

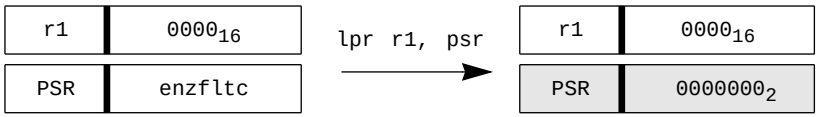
Register	procreg
Processor Status Register	PSR
Interrupt Base Register	INTBASE
Interrupt Stack Pointer	ISP

Refer to “REGISTER SET” on page 2-3 for more information on these registers.

Flags: PSR flags are affected by the values loaded into them. Otherwise, no PSR flags are affected.

Traps: None.

Example: This example loads register PSR from register R1.



LSHi — Logical Shift Integer

LSHB, LSHW

LSHi	***count,***	***dest***
	reg/imm	reg
	read.B	write.i

The **LSHi** instruction performs a logical shift on the ***dest*** operand as specified by the ***count*** operand.

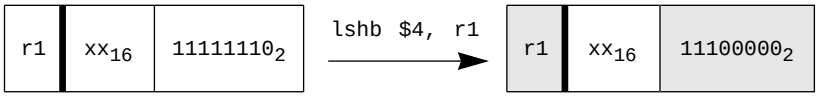
The ***count*** operand is interpreted as a signed integer; the ***dest*** operand is interpreted as an unsigned integer. The sign of ***count*** determines the direction of the shift. A positive ***count*** specifies a left shift; a negative ***count*** specifies a right shift. The absolute value of the ***count*** gives the number of bit positions to shift the ***dest*** operand. The ***count*** operand value must be within the range −7 to +7 if **LSHB** is used, and −15 to +15 if **LSHW** is used; otherwise, the result is unpredictable. All bits shifted out of ***dest*** are lost, and bit positions emptied by the shift are filled with zeros.

Flags: None

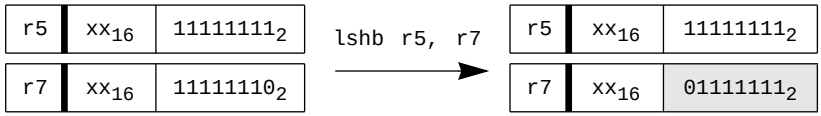
Traps: None

Examples:

1. Example 1 shifts the low order byte of register R1 four bit positions to the left. The remaining bytes of register R1 is unaffected.



2. Example 2 reads a byte from register R5. Based on this value, it shifts the low order byte of register R7. The remaining bytes of register R7 is unaffected.



MOVB, MOVW

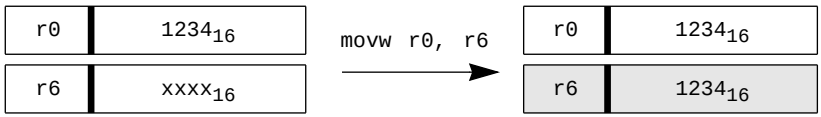
MOVi	***src***,	***dest***
	reg/imm	reg
	read.i	write.i

The **MOVi** instructions copy the ***src*** operand to the ***dest*** register.

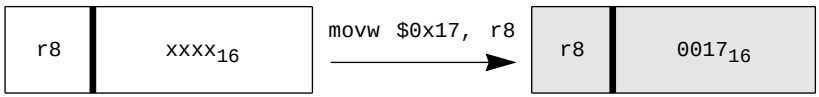
Flags: None

Traps: None

Examples: 1. This example copies the contents of register R0 to register R6.



2. This example sets R8 to the value 17_{16}.



MOVXB*src*, *dest*

reg	reg
read.i	write.W

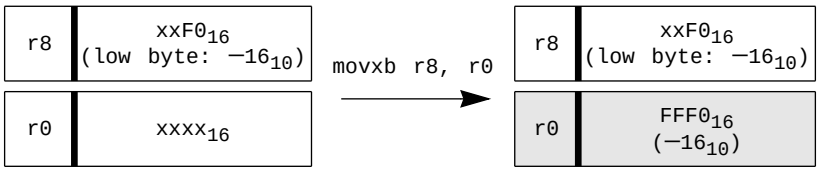
The **MOVXB** instruction converts the signed integer ***src*** operand to the word ***dest*** operand. The sign is preserved through sign-extension.

Flags: None

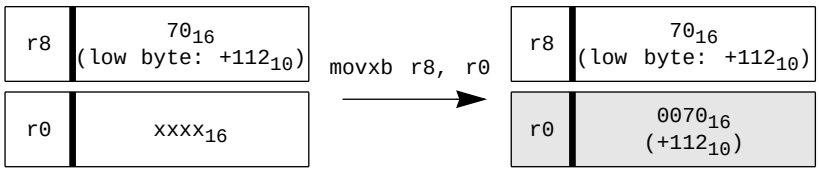
Traps: None

Examples: These examples copy the low order byte of register R8 to the low order byte of register R0, and extend the sign bit of the byte through the next 8 bits of register R0.

1. This example illustrates negative sign extension.



2. This example illustrates positive sign extension.



MOVZB

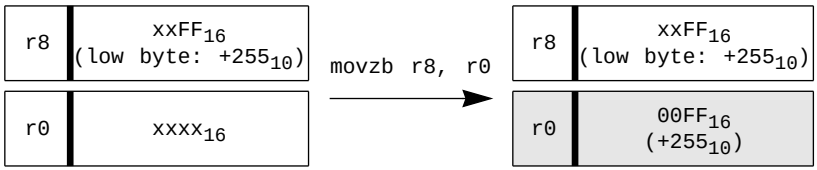
MOVZB*src*,	d***est***
reg	reg
read.i	write.W

The **MOVZB** instruction converts the unsigned integer ***src*** operand to the unsigned word ***dest*** operand. The high order bits are filled with zeros.

Flags: None

Traps: None

Example: This example copies the low order byte of register R8 to the low order byte of register R0, and sets the next 8 bits of register R0 to zero.



MULi — Multiply Integer

MULB, MULW

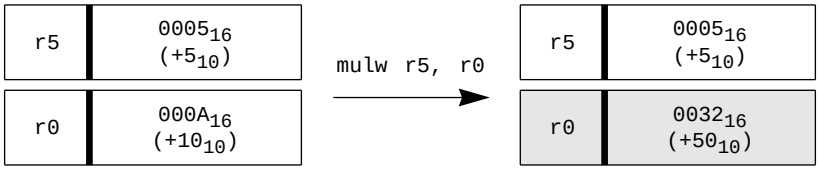
MULI	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **MULi** instructions multiply the ***src*** operand by the ***dest*** operand and places the result in the ***dest*** operand. Both operands are interpreted as signed integers. If the resulting product cannot be represented exactly in the ***dest*** operand, then the high order bits are truncated.

Flags: None

Traps: None

Example: This example multiplies register R5 by R0, and places the result in register R0.



NOP No Operation


```
NOP
```

The **NOP** instruction passes control to the next sequential instruction. No operation is performed.

Flags: None

Traps: None

Example:

```
nop
```

ORB, ORW

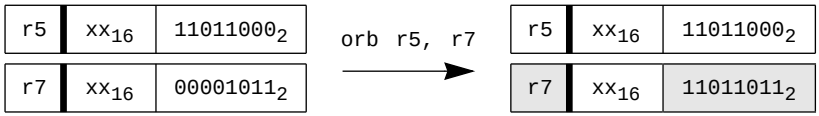
ORi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **ORi** instructions perform a bitwise logical OR operation on the ***src*** and ***dest*** operands, and places the result in the ***dest*** operand.

Flags: None

Traps: None

Example: This example ORs the low order bytes of registers R5 and R7, and places the result in the low order byte of register R7. The remaining byte of register R7 is unaffected.



RETX

The **RETX** instruction returns control from a trap service procedure. The following steps are performed:

1. The instruction pops a 16-bit return address from the interrupt stack, and loads it into bits 1 through 16 of the PC.

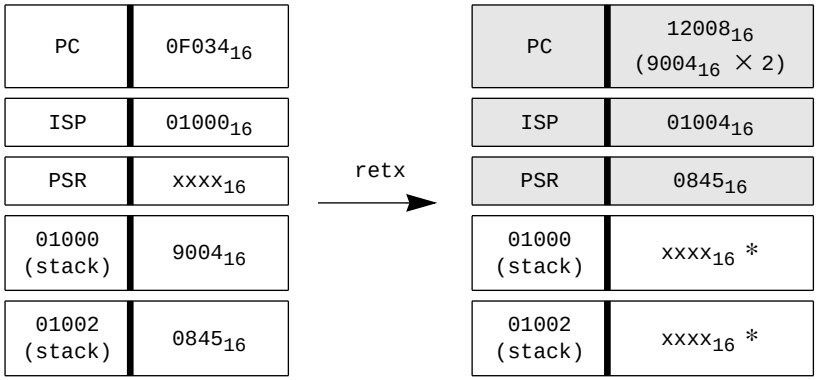
2. The instruction then pops a 16-bit PSR value from the interrupt stack into the PSR.

The **RETX** instruction does not change the contents of memory locations indicated by an asterisk *. However, information that is outside the stack should be considered unpredictable for other reasons.

Flags: All PSR flag states are restored from the stack.

Traps: None

Example: This example returns control from an interrupt service procedure.



Scond — Save Condition as Boolean

SEQ, SNE, SCS, SCC, SHI, SLS, SGT, SLE, SFS, SFC, SLO, SHS, SLT, SGE

Scond ***dest***

reg

write.W

The **Scond** instruction sets the ***dest*** operand to the integer value 1 if the condition specified in **cond** is true, and clears it to 0 if it is false.

cond is a two-character condition code that specifies the state of a flag or flags in the PSR. If the flag(s) are set as required by the specified **cond**, the condition is true; otherwise, the condition is false. The following table describes the possible **cond** codes and the related PSR flag settings:

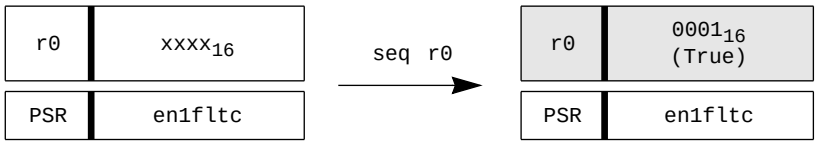
cond Code	**Condition**	**True State**
EQ	Equal	Z flag is 1
NE	Not Equal	Z flag is 0
CS	Carry Set	C flag is 1
CC	Carry Clear	C flag is 0
HI	Higher	L flag is 1
LS	Lower or Same	L flag is 0
GT	Greater Than	N flag is 1
LE	Less Than or Equal	N flag is 0
FS	Flag Set	F flag is 1
FC	Flag Clear	F flag is 0
LO	Lower	Z and L flags are 0
HS	Higher or Same	Z or L flag is 1
LT	Less Than	Z and N flags are 0
GE	Greater Than or Equal	Z or N flag is 1

Flags: None

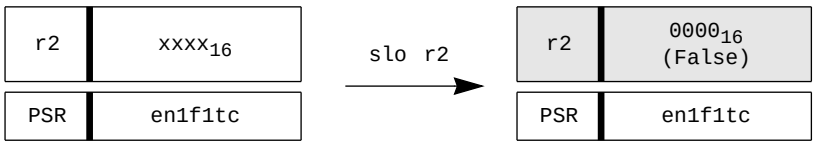
Traps: None

Examples:

1. Example 1 sets register R0 to 1 if the PSR.Z flag is set and to 0 if it is clear.



2. Example 2 sets register R2 to 1 if the PSR.Z and PSR.L flags are clear and to 0 if they are not clear.



SPR	*src,*	*dest*
	procreg	reg
	read.W	write.W

The **SPR** instruction stores the processor register specified by ***src***, in the ***dest*** operand. If ***src*** is INTBASE or ISP, only bits 0 through 15 of ***src*** are stored.

The following processor registers may be stored:

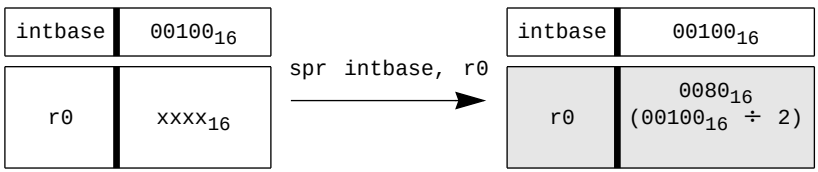
Register	procreg
Processor Status Register	PSR
Interrupt Base Register	INTBASE
Interrupt Stack Pointer	ISP

Refer to “REGISTER SET” on page 2-3 for more information on these registers.

Flags: None

Traps: None.

Example: This example copies the INTBASE register to register R0.



STORi — Store Integer

STORB, STORW

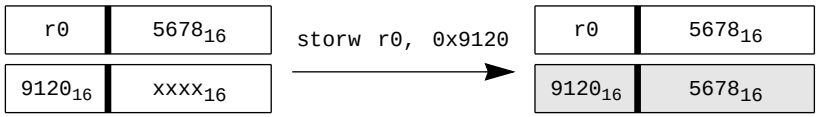
STORi	***src***,	***dest***
	reg	abs/rel/far
	read.i	write.i

The **STORi** instructions store the ***src*** register operand in the ***dest*** memory operand.

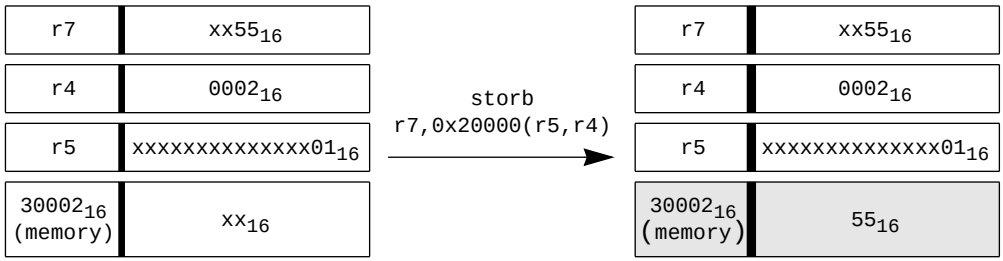
Flags: None

Traps: None

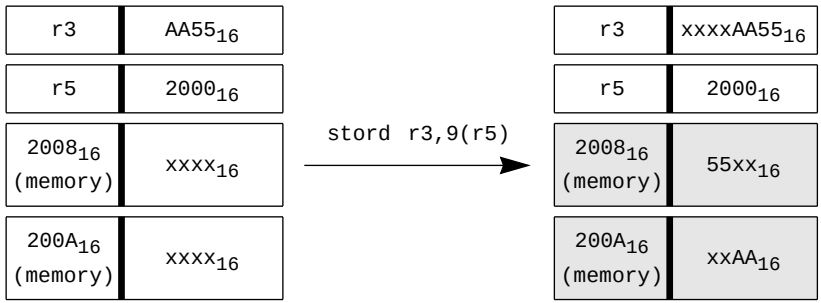
Examples: 1. This example copies the contents of register R0 to the word at address 9120_{16}.



2. Example 2 stores the low order byte from r7 at address 30002_{16}. The address is formed by adding 20000_{16} to the value in R5 concatenated with the value in R4.



3. Example 3 copies the contents of register R3 to the non-aligned word at address 9(R5).



SUBi — Subtract Integer

SUBB, SUBW

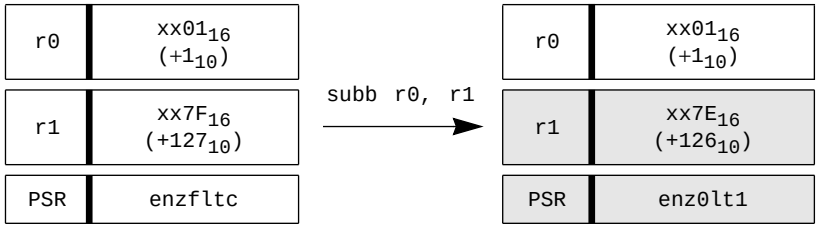
SUBi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **SUBi** instructions subtract the ***src*** operand from the ***dest*** operand, and places the result in the ***dest*** operand.

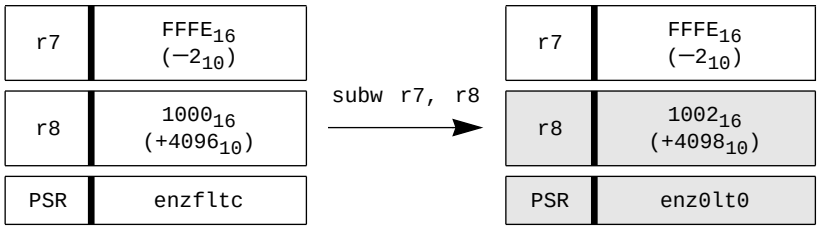
Flags: During execution of a **SUBi** instruction, PSR.C is set to 1 if a borrow occurs, and cleared to 0 if no borrow occurs. PSR.F is set to 1 if an overflow occurs, and cleared to 0 if there is no overflow.

Traps: None

Examples: 1. Example 1 subtracts the low order byte of register R0 from the low order byte of register R1, and places the result in the low order byte of register R1. The remaining byte of register R1 is not affected.



2. Example 2 subtracts the word in register R7 from the word in register R8, and places the result in register R8.



SUBCi — Subtract Integer with Carry

SUBCB, SUBCW

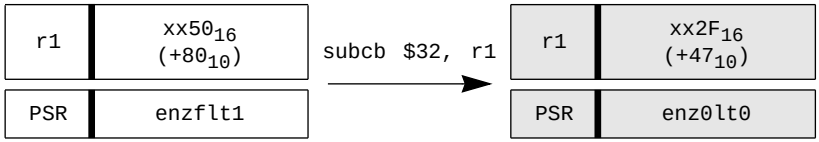
SUBCi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **SUBCi** instructions subtract the sum of the ***src*** operand and the PSR.C flag from the ***dest*** operand, and places the result in the ***dest*** operand.

Flags: PSR.C is set to 1 if a borrow occurs and cleared to 0 if there is no borrow. 0 PSR.F is set to 1 if an overflow occurs and cleared to 0 if there is no overflow.

Traps: None

Example: This example subtracts the sum of 32 and the PSR.C flag value from the low order byte of register R1 and places the result in the low order byte of register R1. The remaining bytes of register R1 is not affected.



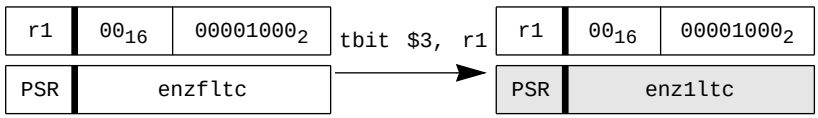
TBIT	***offset,***	***src***
	reg/imm	reg
	read.W	read.W

The **TBIT** instruction copies the bit located in register ***src*** at the bit position specified by ***offset***, to the PSR.F flag. The ***offset*** value must be in the range 0 through 15; otherwise, the result is unpredictable.

Flags: PSR.F is set to the value of the specified bit.

Traps: None

Example: This example copies bit number 3, i.e., the fourth bit from the right, in register R1 to the PSR.F flag.



Flags:

Traps:

Example:

```
WAIT
```

The **WAIT** instruction suspends program execution until an interrupt occurs. An interrupt restores program execution by passing it to an interrupt service procedure. When the **WAIT** instruction is interrupted, the return address saved on the stack is the address of the instruction following the **WAIT** instruction.

Flags: None

Traps: None

Example:

```
wait
```

XORi — Bitwise Logical Exclusive OR

XORB, XORW

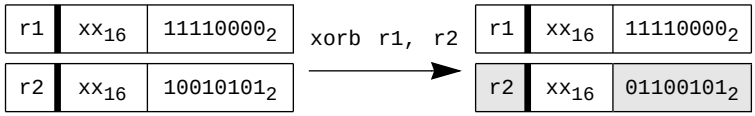
XORi	***src***,	***dest***
	reg/imm	reg
	read.i	rmw.i

The **XORi** instructions perform a bitwise logical exclusive OR operation on the ***src*** and ***dest*** operands, and places the result in the ***dest*** operand.

Flags: None

Traps: None

Example: This example XORs the low order bytes of registers R1 and R2, and places the result in the low order byte of register R2. The remaining byte of R2 is unaffected.



Appendix A
INSTRUCTION SET ENCODING

This appendix describes instruction encoding. Most instructions are encoded using one of the basic instruction formats. Where formats for instructions differ from the basic formats, e.g., load and store instructions, branch instructions and jump instructions, those differences are described separately.

Tables at the end of this Appendix summarize this instruction encoding information.

A.1 INTRODUCTION

Instructions may have zero, one or two operands and are encoded using two or four bytes. All instructions must be word-aligned.

Figure A-1 shows the basic structure of a two-operand instruction.

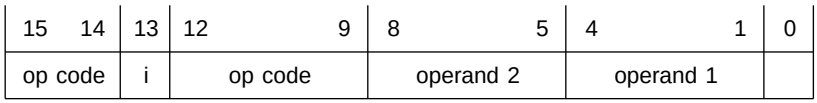


Figure A-1. Basic Instruction Structure

Two or three bits code the operation (op code in bits 14, 15 and sometimes bit 0). One bit indicates the operation length (i in bit 13). Four bits (bits 9 through 12) may further specify the operation or be used for a displacement value. Eight bits (bits 1 through 4 and bits 5 through 8) specify two instruction operands.

Bit 0 Bit 0 is used to extend other fields, e.g., op code or the first operand. See each format for more details.

Bits 1-4 When bits 1 through 4 (operand 1) specify a general purpose register, it is usually the source register. This field may also contain a vector, a constant (immediate) value or a displacement value. If the constant or displacement value does not fit in the space allotted to it (its length is medium), it may be encoded in the next two bytes. See the format descriptions that follow for details.

Bits 5-8 When bits 5 through 8 (operand 2) specify a general purpose register, it is usually the destination register. This field may also specify other special instruction options, such as a condition or a dedicated processor register, depending on the instruction.

Bits 9-12 Bits 9 through 12 may contain the operation code and/or a displacement value.

Bit 13 Bit 13 indicates the integer operation length (i). If i = 0, it is a byte (8-bit) operation; if i = 1, it is a word (16-bit) operation.

Bits 14-15 Bits 14 and 15, and sometimes bit 0, specify an operation code. Often, other bits are used with this op code to further specify the operation.

A.2 INSTRUCTION FORMATS

Most instructions use one of the basic formats described in the next section. In addition, load and store instructions, branch instructions (**BR**, **Bcond** and **BAL**) and jump instructions (**JUMP**, **Jcond** and **JAL**) each use a different format. These formats are all described in the sections that follow.

A.2.1 Basic Instruction Formats

The **ADDi**, **ADDCi**, **ADDUi**, **ANDi**, **ASHUi**, **CMPi**, **LSHi**, **MOVi**, **MULi**, **ORi**, **SUBi**, **SUBCi**, **TBIT** and **XORi** instructions use one of the basic formats described in this section. The format used depends on the operands.

Register to register operations Figure A-2 shows the format for instructions with two general purpose register operands.

15	14	13	12 9	8 5	4 1	0
0	1	i	op code	dest reg	src reg	1

Figure A-2. Register to Register Format

Short immediate to register operations A short immediate value is one that fits in the space provided in a 2-byte basic instruction format. The value −16 and all values in the range −14 through 15 can be encoded in this format.

Figure A-3 shows the basic format for instructions that have short immediate values as operands. The core sign-extends the value in bits 0 through 4 (imm) to form a 16-bit immediate operand.

15 14	13	12 9	8 5	4 0
0 0	i	op code	dest reg	imm

Figure A-3. Short Immediate Value to Register Format

Medium immediate to register operations

An immediate value that does not fit in the space allocated for the first operand in a 2-byte format, as shown in Figure A-3, is a medium value. The signed 16-bit medium value is placed (encoded) in the next two bytes. All values in the range -32768 through 32767 can be encoded in this format.

Figure A-4 shows the basic format, when the first operand is a medium immediate value (imm).

31 16	15 14	13	12 9	8 5	4 0
imm	0 0	i	op code	dest reg	1 0 0 0 1

Figure A-4. Medium Immediate Value to Register Format

Instructions with special or no operands

The **DI**, **EI**, **EXCP**, **LPR**, **MOVXB**, **MOVZB**, **RETX**, **Scond**, **SPR** and **WAIT** instructions either have unique operands or no operands at all. These instructions use the format illustrated in Figure A-5.

15 14	13	12 9	8 5	4 1	0
0 1	1	op code	procreg/cond/reg	reg/vector	0

Figure A-5. Format for Instructions with Special or No Operands

A.2.2 Load and Store Instructions

The **LOADi** and **STORi** instructions use the same formats. However, their op codes in bits 14 and 15 differ, and they specify different registers (reg in bits 5 through 8).

For the **LOADi** instruction, the op code (bits 14 and 15) is 10 and the reg field identifies the destination register. For the **STORi** instruction, the op code is 11 and the reg field identifies the source register. See Table A-1.

Table A-1. Coding for Load and Store Op code and Register Type

Instruction	op code	reg
Load	1 0	Destination Register
Store	1 1	Source Register

The format to use for load and store instructions depends on the addressing mode and on the length of their displacement values. See also “Addressing Modes” on page 2-9.

Relative Addressing Mode

Short Displacement Values

A short displacement value fits within the field allotted for the displacement value in a 2-byte format (bit 0 and bits 9 through 12). This applies to any odd or even displacement value in the range 0 through 15, or any even displacement value in the range 16 through 30.

During execution, the core zero-extends the displacement field to 18 bits.

Figure A-6 shows the format for load and store instructions, when the displacement value is short.

15 14	13	12 9	8 5	4 1	0
op code	i	disp (d_4–d_1)	reg	base reg	d_0

Figure A-6. Load/Store Format, Relative with Short Displacement Value

Medium Displacement Values

A medium displacement value is one that does not fit in a 2-byte format. In this case, the 18-bit displacement value is encoded by using two additional bytes.

Figure A-7 shows the format for load and store instructions when the addressing mode is relative and the displacement value is medium.

31 16	15 14	13	12 11	10 9	8 5	4 1	0
disp (d_{15}–d_0)	op code	i	1 0	d_{17} d_{16}	reg	base reg	1

Figure A-7. Load/Store Format, Relative with Medium Displacement Value

Far- Relative Addressing Mode

The addressing mode of load and store instructions is called far-relative when the base address is specified by a pair of adjacent registers. See Figure A-8.

The base pair field may encode any general purpose register except SP, i.e., R0-RA that contains the 16 least significant bits of the base address. The two most significant bits of the base address are taken from the next consecutive register. In this case, the entire 18-bit displacement value is encoded by using two additional bytes.

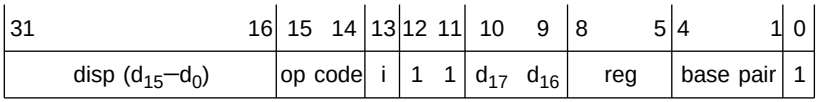


Figure A-8. Load/Store Format, Far-Relative

Absolute Addressing Mode

Figure A-9 shows the format for load and store instructions when the addressing mode is absolute.

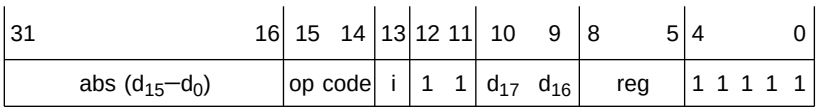


Figure A-9. Load/Store Format, Absolute

A.2.3 Branch Instructions

Branch instructions, i.e., **Bcond**, **BAL** and **BR**, specify the target address as a displacement from the address currently in the Program Counter (PC). The displacement value is interpreted as a signed integer. The target address is the value in the displacement field plus the address currently in the PC.

Since all instructions are word-aligned, in the **Bcond**, **BAL** and **BR** instructions, displacement values must be even.

BR and Bcond Instructions

The **BR** and **Bcond** instructions can be encoded in 2 or 4 bytes, depending on whether the displacement value is short or medium, respectively. The core sign-extends short and medium branch displacement values to 18 bits.

In the **Bcond** instruction, bits 5 through 8 specify the condition.

Short Displacement Values

Even displacement values in the range -256 through 254 are called short. Short values fit in nine bits provided in a 2-byte format. The displacement value is encoded in bits 0 through 4 (d_0 through d_4, respectively) and bits 9 through 12 (d_5 through d_8, respectively).

Figure A-10 shows the format of **BR** or **Bcond** instructions with short displacement values. Note that in this format, bit 0 (d_0) must be 0.

15 14	13	12 9	8 5	4 0
0 1	0	disp (d_8–d_5)	cond	disp (d_4–d_0)

Figure A-10. BR or Bcond Format, Short Displacement Value

Medium Displacement Values

When the displacement value does not fit in the 2-byte format, it is called medium and encoded into two additional bytes as shown in Figure A-11. Note that bit 16 (d_0) must be 0.

31 16	15 14	13	12 9	8 5	4	3 0
disp (d_{15}–d_0)	0 0	0	1 0 1 0	cond	d_{16}	1 1 1 0

Figure A-11. BR or Bcond Format, Medium Displacement Value

The BAL Instruction

The BAL Instruction

For the **BAL** instruction, the displacement value is encoded as shown in Figure A-12. Bits 5 through 8 specify the link register. As in **BR** and **Bcond** instructions, bit 16 (d_0) must be 0.

31 16	15 14	13	12 9	8 5	4	3 0
disp (d_{15}–d_0)	0 0	1	1 0 1 0	link reg	d_{16}	1 1 1 0

Figure A-12. BAL Format

A.2.4 Jump Instructions

The JUMP and Jcond Instructions

JUMP and **Jcond** instructions use the format shown in Figure A-13.

15 14	13	12 9	8 5	4 1	0
0 1	0	1 0 1 0	cond	target reg	1

Figure A-13. JUMP and Jcond Instruction Format

The JAL Instruction

In the **JAL** instruction, bits 5 through 8 specify the link register. It is encoded as shown in Figure A-14.

15 14	13	12 9	8 5	4 1	0
0 1	1	1 0 1 0	link reg	target reg	1

Figure A-14. JAL Instruction Format

A.3 UNDEFINED OP CODES

The following op codes cause an undefined instruction trap.

Table A-2. Undefined Op Codes

Op Code	i	Op Code	Operand 2	Operand 1	Op Code or Operand 1	Comment
0 0	0	1 0 1 1	X X X X	X X X X	X	**TBIT** on a byte instead of a word
0 0	X	1 0 1 0	X X X X	X X X X	1	**Bcond, BR** or **BAL** with odd target
0 0	X	0 0 1 0	X X X X	X X X X	X	reserved
0 1	X	0 0 1 0	X X X X	X X X X	1	reserved
0 1	1	0 1 1 0	X X X X	X X X X	0	reserved

Op Code	i	Op Code	Operand 2	Operand 1	Op Code or Operand 1	Comment
0 1	1	0 0 X X	X X X X	X X X X	0	reserved
0 1	1	1 0 1 X	X X X X	X X X X	0	reserved
0 1	0	1 0 1 1	X X X X	X X X X	1	**TBIT** on a byte instead of a word

A.4 CR16A INSTRUCTION SET SUMMARY

Table A-3. Notation Conventions for Instruction Set Summary

i = Operation length field
0 – Byte (8 bits)
1 – Word (16 bits)

abs = Absolute address

imm = Immediate value

disp = Displacement value

dest = Destination

src = Source

Rsrc, Rdest, Rlink, Rbase, Rpair, Rtarget, Roffset = Source, destination, link, base, base pair, target or offset register, respectively.

0000 – R0
0001 – R1
0010 – R2
0011 – R3
0100 – R4
0101 – R5
0110 – R6
0111 – R7
1000 – R8
1001 – R9
1010 – R10
1011 – R11
1100 – R12
1101 – R13
1110 – RA
1111 – SP

Rproc =

Dedicated CPU register
0001 – PSR
0011 – INTBASE
1011 – ISP

cond =

Condition code field

0000 – EQ	Equal	Z = 1
0001 – NE	Not Equal	Z = 0
1101 – GE	Greater than or Equal	N = 1 or Z = 1
0010 – CS	Carry Set	C = 1
0011 – CC	Carry Clear	C = 0
0100 – HI	Higher than	L = 1
0101 – LS	Lower than or the Same as	L = 0
1010 – LO	Lower than	L = 0 and Z = 0
1011 – HS	Higher than or the Same as	L = 1 or Z = 1
0110 – GT	Greater Than	N = 1
0111 – LE	Less than or Equal	N = 0
1000 – FS	Flag Set	F = 1
1001 – FC	Flag Clear	F = 0
1100 – LT	Less Than	N = 0 and Z = 0

vector =

Exception vector (used by EXCP instruction)
0101 – SVC
0110 – DVZ
0111 – FLG
1000 – BPT
1010 – UND
others– reserved

Table A-4. Instruction Encoding

Mnemonic	Operands	15	14	13	12			9	8	5	4	1	0
MOVES													
MOVi	Rsrc, Rdest	0	1	i	1	1	0	0	Rdest		Rsrc		1
	imm, Rdest	0	0	i	1	1	0	0	Rdest		imm		i_0
MOVXB	Rsrc, Rdest	0	1	1	0	1	0	0	Rdest		Rsrc		0
MOVZB	Rsrc, Rdest	0	1	1	0	1	0	1	Rdest		Rsrc		0
INTEGER ARITHMETIC													
ADDi	Rsrc, Rdest	0	1	i	0	0	0	0	Rdest		Rsrc		1
	imm, Rdest	0	0	i	0	0	0	0	Rdest		imm		i_0
ADDUi	Rsrc, Rdest	0	1	i	0	0	0	1	Rdest		Rsrc		1
	imm, Rdest	0	0	i	0	0	0	1	Rdest		imm		i_0
ADDCi	Rsrc, Rdest	0	1	i	1	0	0	1	Rdest		Rsrc		1
	imm, Rdest	0	0	i	1	0	0	1	Rdest		imm		i_0
MULi	Rsrc, Rdest	0	1	i	0	0	1	1	Rdest		Rsrc		1
	imm, Rdest	0	0	i	0	0	1	1	Rdest		imm		i_0
SUBi	Rsrc, Rdest	0	1	i	1	1	1	1	Rdest		Rsrc		1
	imm, Rdest	0	0	i	1	1	1	1	Rdest		imm		i_0
SUBCi	Rsrc, Rdest	0	1	i	1	1	0	1	Rdest		Rsrc		1
	imm, Rdest	0	0	i	1	1	0	1	Rdest		imm		i_0
INTEGER COMPARISON													
CMPi	Rsrc, Rdest	0	1	i	0	1	1	1	Rdest		Rsrc		1
	imm, Rdest	0	0	i	0	1	1	1	Rdest		imm		i_0
LOGICAL AND BOOLEAN													
ANDi	Rsrc, Rdest	0	1	i	1	0	0	0	Rdest		Rsrc		1
	imm, Rdest	0	0	i	1	0	0	0	Rdest		imm		i_0
ORi	Rsrc, Rdest	0	1	i	1	1	1	0	Rdest		Rsrc		1
	imm, Rdest	0	0	i	1	1	1	0	Rdest		imm		i_0
Scond	Rdest	0	1	1	0	1	1	1	cond		Rdest		0
XORi	Rsrc, Rdest	0	1	i	0	1	1	0	Rdest		Rsrc		1
	imm, Rdest	0	0	i	0	1	1	0	Rdest		imm		i_0
SHIFTS													
ASHUi	Rsrc, Rdest	0	1	i	0	1	0	0	Rdest		Rsrc		1
	imm, Rdest	0	0	i	0	1	0	0	Rdest		imm		i_0
LSHi	Rsrc, Rdest	0	1	i	0	1	0	1	Rdest		Rsrc		1
	imm, Rdest	0	0	i	0	1	0	1	Rdest		imm		i_0
BITS													
TBIT	Roffset, Rsrc	0	1	1	1	0	1	1	Rsrc		Roffset		1
	imm, Rsrc	0	0	1	1	0	1	1	Rsrc		imm		i_0

Mnemonic	Operands	15	14	13	12			9	8			5	4			1	0

PROCESSOR REGISTER MANIPULATION

LPR	Rsrc, Rproc	0	1	1	1	0	0	0	Rproc				Rsrc				0
SPR	Rproc, Rdest	0	1	1	1	0	0	1	Rproc				Rdest				0

JUMPS AND LINKAGE

Bcond	disp	0	1	0	d_8	d_7	d_6	d_5	cond			d_4	d_{16}	d_3	d_2	d_1	0
		0	0	0	1	0	1	0	cond					1	1	1	0
BAL	Rlink, disp	0	0	1	1	0	1	0	Rlink			d_{16}		1	1	1	0
BR	disp	0	1	0	d_8	d_7	d_6	d_5	1	1	1	0	d_4	d_{16}	d_3	d_2	d_1	0
		0	0	0	1	0	1	0	1	1	1	0			1	1	1	0
EXCP	vector	0	1	1	1	1	0	1	1	1	1	1	vector					0
Jcond	Rtarget	0	1	0	1	0	1	0	cond				Rtarget					1
JAL	Rlink, Rtarget	0	1	1	1	0	1	0	Rlink				Rtarget					1
JUMP	Rtarget	0	1	0	1	0	1	0	1	1	1	0	Rtarget					1
RETX		0	1	1	1	1	0	0	1	1	1	1	1		1	1	1	0

LOAD AND STORE

LOADi	disp(Rbase), Rdest	1	0	i	d_4	d_3	d_2	d_1	Rdest				Rbase				d_0
	disp(Rbase), Rdest	1	0	i	1	0	d_{17}	d_{16}	Rdest				Rbase				1
	disp(Rpair+1, Rpair), Rdest	1	0	i	1	1	d_{17}	d_{16}	Rdest				Rpair				1
	abs, Rdest	1	0	i	1	1	d_{17}	d_{16}	Rdest				1		1	1	1
STORi	Rsrc, disp(Rbase)	1	1	i	d_4	d_3	d_2	d_1	Rsrc				Rbase				d_0
	Rsrc, disp(Rbase)	1	1	i	1	0	d_{17}	d_{16}	Rsrc				Rbase				1
	Rsrc, disp(Rpair+1, Rpair)	1	1	i	1	1	d_{17}	d_{16}	Rsrc				Rpair				1
	Rsrc, abs	1	1	i	1	1	d_{17}	d_{16}	Rsrc				1		1	1	1

MISCELLANEOUS

DI		0	1	1	1	1	1	0	1	1	1	0	1		1	1	1	0
EI		0	1	1	1	1	1	0	1	1	1	1	1		1	1	1	0
NOP		0	0	0	0	0	0	1	0	0	0	0	0		0	0	0	0
WAIT		0	1	1	1	1	1	1	1	1	1	1	1		1	1	1	0

Appendix B
CR16 INSTRUCTION SET

The following instructions are included in the CR16A:

Mnemonic	Operands	Description	Flag
MOVES			
MOVi	Rsrc/imm, Rdest	Move	
MOVXB	Rsrc, Rdest	Move with sign extension	
MOVZB	Rsrc, Rdest	Move with zero extension	
INTEGER ARITHMETIC			
ADDi	Rsrc/imm, Rdest	Add	CF
ADD[U]i	Rsrc/imm, Rdest	Add	
ADDCi	Rsrc/imm, Rdest	Add with carry	CF
MULi	Rsrc/imm, Rdest	Multiply	
SUBi	Rsrc/imm, Rdest	Subtract (Rdest := Rdest − Rsrc)	CF
SUBCi	Rsrc/imm, Rdest	Subtract with carry (Rdest := Rdest − Rsrc − PSR.C)	CF
INTEGER COMPARISON			
CMPi	Rsrc/imm, Rdest	Compare (Rdest − Rsrc)	ZNL
LOGICAL AND BOOLEAN			
ANDi	Rsrc/imm, Rdest	Logical AND	
ORi	Rsrc/imm, Rdest	Logical OR	
Scond	Rdest	Save condition code as boolean	
XORi	Rsrc/imm, Rdest	Logical exclusive OR	
SHIFTS			
ASHUi	Rsrc/imm, Rdest	Arithmetic left/right shift	
LSHi	Rsrc/imm, Rdest	Logical left/right shift	
BITS			
TBIT	Roffset/imm, Rsrc	Test bit	
PROCESSOR REGISTER MANIPULATION			
LPR	Rsrc, Rproc	Load processor register	CTLFZNEPI
SPR	Rproc, Rdest	Store processor register	

Mnemonic	Operands	Description	Flag
JUMPS AND LINKAGE			
Bcond	disp	Conditional branch	
BAL	Rlink, disp	Branch and link	
BR	disp	Branch	
EXCP	vector	Trap (vector)	
Jcond	Rtarget	Conditional Jump	
JAL	Rlink, Rtarget	Jump and link	
JUMP	Rtarget	Jump	
RETX		Return from exception	
LOAD AND STORE			
LOADi	disp(Rbase), Rdest	Load (register relative)	
	disp(Rpair+1, Rpair), Rdest	Load (far-relative)	
	abs, Rdest	Load (absolute)	
STORi	Rsrc, disp(Rbase)	Store (register relative)	
	Rsrc, disp(Rpair +1, Rpair)	Store (far-relative)	
	Rsrc, abs	Store (absolute)	
MISCELLANEOUS			
DI		Disable maskable interrupts	E
EI		Enable maskable interrupts	E
NOP		No operation	
WAIT		Wait for interrupt	

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z